C. Tuchie

Bell Laboratories

UNIX PROGRAMMER'S MANUAL

Program Generic PG-1C300 Issue 3

Published by the UNIX Support Group

March, 1977

UNIX PROGRAMMER'S MANUAL

Program Generic PG-1C300 Issue 3

Published by the UNIX Support Group

March, 1977

This manual was photocomposed in the Murray Hill Computation Center. The text of the manual was prepared using the UNIX *ed* text editor and *troff* formatting program.

PREFACE

This document is published as part of the UNIX Operating System Program Generic, PG-1C300 Issue 3. The development of the Program Generic is the result of the efforts of the members of the Small Systems Development Department (8234).

Most of the commands and system software were written by the Computing Science Research Center (127), especially K. Thompson and D. M. Ritchie. This manual is based on the UNIX PROGRAMMER'S MANUAL, Sixth Edition, May, 1975 by K. Thompson and D. M. Ritchie.

For corrections and comments please contact I. A. Hahner, MH 2F-219, Extension 2771.

INTRODUCTION TO THIS MANUAL

This manual gives descriptions of the publicly available features of UNIX. It provides neither a general overview — see "The UNIX Time-sharing System" (Comm. ACM **17** 7, July 1974, pp. 365-375) for that — nor details of the implementation of the system, which remain to be disclosed.

Within the area it surveys, the manual attempts to be as complete and timely as possible. A conscious decision was made to describe each program in exactly the state it was in at the time its manual section was prepared. In particular, the desire to describe something as it should be, not as it is, was resisted. Inevitably, this means that many sections will soon be out of date.

This manual is divided into eight sections:

I. Commands
II. System Calls
III. Subroutines
IV. Drivers
V. File Formats
VI. User Programs
VII. Tables
VIII. System Programs

Commands are programs intended to be invoked directly by the user, in contradistinction to subroutines, which are intended to be called by the user's programs. Commands generally reside in directory */bin* (for *bin*ary programs). Some programs also reside in */usr/bin,* to save space in */bin.* These directories are searched automatically by the command interpreter.

System calls are entries into the UNIX supervisor. In assembly language, they are coded with the use of the opcode *sys,* a synonym for the *trap* instruction. In this edition, the C language interface routines to the system calls have been incorporated in section II.

A small assortment of subroutines is available; they are described in section III. The binary form of most of them is kept in the system library */lib/liba.a.* The subroutines available from C and from Fortran are also included; they reside in */lib/libc.a* and */lib/libf.a* respectively.

Drivers (section IV) discusses the characteristics of each system "file" which actually refers to an I/O device. The names in this section refer in most cases to the DEC device names for the hardware, instead of the names of the special files themselves.

File Formats (section V)documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler's intermediate files.

User Programs (section VI), while part of the Standard UNIX system, are not fully supported, and the principal reason for listing them is to indicate their existence without necessarily giving a complete description.

Section VII groups together the information pertaining to tabular data.

Section VIII discusses commands which are not intended for use by the ordinary user, in some cases because they disclose information in which he is presumably not interested, and in others because they perform privileged functions.

Each section consists of a number of independent entries of one or more pages. Below the program application heading is the name of the entry in bold-face type. Entries within each section are alphabetized. The page numbers of each entry start at 1.

All entries are based on a common format, not all of whose subsections will always appear.

> The *name* section repeats the entry name and gives a very short description of its purpose.
>
> The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands section:
>
> > **Boldface** words are considered literals, and are typed just as they appear.
> >
> > Square brackets ([ ]) around an argument indicate that the argument is optional. When an argument is given as "name", it always refers to a file name.
> >
> > Ellipses "..." are used to show that the previous argument-prototype may be repeated.
> >
> > A final convention is used by the commands themselves. An argument beginning with a minus sign "–" is often taken to mean some sort of flag argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with "–".
>
> The *description* section discusses in detail the subject at hand.
>
> The *files* section gives the names of files which are built into the program.
>
> A *see also* section gives pointers to related information.
>
> A *diagnostics* section discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.
>
> The *bugs* section gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

At the beginning of this document is a table of contents, organized by section and alphabetically within each section. There is also a permuted index derived from the table of contents. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands which exist only to exercise a particular system call.

This manual was prepared using the UNIX text editor *ed* and the formatting program *troff.*

HOW TO GET STARTED

This section provides the basic information you need to get started on UNIX: how to log in and log out, how to communicate through your terminal, and how to run a program. See "UNIX for Beginners" by Brian W. Kernighan (PA-1C3019) for a more complete introduction to the system.

Logging in. You must call UNIX from an appropriate terminal. UNIX supports full-duplex ASCII terminals typified by the TTY 37, the GE Terminet 300, the Dasi 300, and various graphical terminals. UNIX also supports the half-duplex TTY Model 40/1 and other terminals that can operate in half-duplex mode such as the TTY Model 40/2, the Tektronix 4010 and 4012. You must also have a valid user name, which may be obtained, together with the telephone number, from the system administrators. The same telephone number serves terminals operating at all the standard speeds (110, 150, and 300 baud). The Model 40/1 and other terminals which operate at 1200 baud require a different number. After a data connection is established, the login procedure depends on what kind of terminal you are using.

> *300-baud terminals:* Such terminals include the GE Terminet 300, most display terminals, Execuport, TI, GSI, Teletype Models 40/2 and 43, and certain Anderson-Jacobson terminals. These terminals generally have a speed switch which should be set at "300" (or "30" for 30 characters per second) and a half/full duplex switch which should be set at full-duplex. (This switch will often have to be changed since many other systems require half-duplex). When a connection is established, the system types "login:"; you type your user name, followed by the "return" key. If you have a password, the system asks for it and turns off the printer on the terminal so the password will not appear. After you have logged in, the "return", "new line", or "linefeed" keys will give exactly the same results.
>
> *TTY 37 terminal:* When you have established a data connection, the system types out a few garbage characters (the "login:" message at the wrong speed). Depress the "break" (or "interrupt") key; this is a speed-independent signal to UNIX that a 150-baud terminal is in use. The system then will type "login:," this time at the correct speed; you respond with your user name. From the TTY 37 terminal, and any other which has the "new-line" function (combined carriage return and linefeed), terminate each line you type with the "new-line" key (*not* the "return" key).
>
> *TTY Model 40/1:* When a connection is established, the system types "login:"; you type your user name, followed by the "return" key. This terminal transmits a line-at-a-time rather than a character-at-a-time. When the return (not newline) key is depressed all characters from the beginning of the line up to and including the carriage return are transmitted. Because of this an extra newline is sent to the Model 40 after each transmission to place the cursor at the beginning of a line so that the prompt character or any other characters sent by UNIX to the terminal will not be included in the user's next transmission. The newline delay (nl3) is set to allow for the line feed delay on the Model 40/1 printer. If a printer is not in use, setting the delay to nl0 using the stty command (stty (I)) will speed up the character output rate considerably.

For all these terminals, it is important that you type your name in lower-case if possible; if you type upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent upper-case letters to lower case.

The evidence that you have successfully logged in is that the Shell program will type a "%" to you. (The Shell is described below under "How to run a program.")

For more information, consult *getty* (VIII), which discusses the login sequence in more detail, *tty* (IV), which discusses typewriter I/O, and *kl* (IV),which discusses the console typewriter.

Logging out. There are three ways to log out:

You can simply hang up the phone.

You can log out by typing an end-of-file indication (EOT character, control "d") to the Shell. The Shell will terminate and the "login: " message will appear again.

You can also log in directly as another user by giving a *login* command (I).

How to communicate through your terminal. When you type to UNIX, the characters are collected and saved by the system. The characters will not be given to a program until you type a return (or new-line), as described above in *Logging in.*

Except when using the Model 40/1, UNIX typewriter I/O is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is typing at you. Of course, if you type during output, the output will have the input characters interspersed. However, whatever you type will be saved up and interpreted in correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away all the saved characters. When using a Model 40/1, a line can be transmitted to UNIX at any time that UNIX is not sending to the Model 40/1. A drawback to trying this when you are expecting output is that UNIX may have output ready for you while you are in the process of preparing a line of input. In this case UNIX will send the output and the characters will be displayed starting at the current cursor position, corrupting the line being prepared.

On a typewriter input line, the character "@" kills all the characters typed before it, so typing mistakes can be repaired on a single line. Also, the character "#" erases the last character typed. Successive uses of "#" erase characters back to, but not beyond, the beginning of the line. "@" and "#" can be transmitted to a program by preceding them with "\". (So, to erase "\", you need two "#"s). The kill "@" and erase "#" can be used with a Model 40/1 but there are alternatives since transmission is line-at-a-time. Prior to sending a line, the cursor positioning keys in combination with the character delete and insert keys or just overwriting can be used to correct a line.

The ASCII SOH (control-A) character can be typed to temporarily stop output. This is especially useful with CRT terminals to let you read the output before it disappears from the screen. Output is resumed when a second SOH or any other character is typed. SOH characters typed in this regard are not passed to programs.

The ASCII "delete" (a.k.a. "rubout") character is not passed to programs but instead generates an *interrupt signal.* This signal generally causes whatever program you are running to terminate. It is typically used to stop a long printout that you don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). The editor, for example, catches interrupts and stops what it is doing, instead of terminating, so that an interrupt can be used to halt an editor printout without losing the file being edited. When using a Model 40/1, the delete character (or any character for that matter) cannot be sent to UNIX while UNIX is sending output to the terminal. To cause an interrupt signal when using a Model 40/1, the intrpt key is used to stop the output. The key is depressed once which lights the key top light; it must be depressed again to extinguish the light. When the light is extinguished the output will be stopped temporarily. A "delete" character may be entered at this time or the output may be continued by depressing the intrpt key two more times. If the light is not extinguished within about 6 seconds, the system will hang up.

The *quit* signal is generated by typing the ASCII FS character. It not only causes a running program to terminate but also generates a file with the core image of the terminated process. Quit is useful for debugging. When using a Model 40/1, the procedure for entering a quit signal is similar to that for entering the "delete" character.

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about whether you have a terminal with the new-line function or whether it must be simulated with carriage-return and line-feed. In the latter case, all input carriage returns are turned to new-line characters (the standard line delimiter) and both a carriage return and a line feed are echoed to the terminal. If you get into the wrong mode, the *stty* command (I) will rescue you.

Tab characters are used freely in UNIX source programs. If your terminal does not have the tab function, you can arrange to have them turned into spaces during output, and echoed as spaces during input. The system assumes that tabs are set every eight columns. Again, the *stty* command (I) will set or reset this mode. Also, there is a file which, if printed on TTY 37 or Terminet 300 terminals, will set the tab stops correctly (*tabs* (VII)).

How to run a program; the Shell. When you have successfully logged into UNIX, a program called the Shell is listening to your terminal. The Shell reads typed-in lines, splits them up into a command name and arguments, and executes the command. A command is simply an executable program. The Shell looks first in your current directory (see next section) for a program with the given name, and if none is there, then in a system directory. There is nothing special about system-provided commands except that they are kept in a directory where the Shell can find them.

The command name is always the first word on an input line; it and its arguments are separated from one another by spaces.

When a program terminates, the Shell will ordinarily regain control and type a "%" at you to indicate that it is ready for another command.

The Shell has many other capabilities, which are described in detail in section *sh* (I).

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he also created a directory for you (ordinarily with the same name as your user name). When you log in, any file name you type is by default in this directory. Since you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. Permissions to have your will with other directories and files will have been granted or denied to you by their owners. As a matter of observed fact, few UNIX users protect their files from destruction, let alone perusal, by other users.

To change the current directory (but not the set of permissions you were endowed with at login) use *chdir* (I).

Path names. To refer to files not in the current directory, you must use a path name. Full path names begin with "/", the name of the root directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a "/") until finally the file name is reached. E.g.: */usr/lem/filex* refers to the file *filex* in the directory *lem; lem* is itself a subdirectory of *usr; usr* springs directly from the root directory.

If your current directory has subdirectories, the path names of files therein begin with the name of the subdirectory (no prefixed "/").

Without important exception, a path name may be used anywhere a file name is required.

Important commands which modify the contents of files are *cp* (I), *mv* (I), and *rm* (I), which respectively copy, move (i.e. rename) and remove files. To find out the status of files or directories, use *ls* (I). See *mkdir* (I) for making directories; *rmdir* (I) for destroying them.

For a fuller discussion of the file system, see "The UNIX Time-Sharing System," by K. Thompson and D. M. Ritchie (PD-1C300 section 1). It may also be useful to glance through section II of this manual, which discusses system calls, even if you don't intend to deal with the system at that level.

Writing a program. To enter the text of a source program into a UNIX file, use *ed* (I). The three principal languages in UNIX are assembly language (see *as* (I)), Fortran (see *fc* (I)), and C (see *cc* (I)). After the program text has been entered through the editor and written on a file, you can give the file to the appropriate language processor as an argument. The output of the language processor will be left on a file in the current directory named "a.out". (If the output is precious, use *mv* to move it to a less exposed name soon.) If you wrote in assembly language, you will probably need to load the program with library subroutines; see *ld* (I). The other two language processors call the loader automatically.

When you have finally gone through this entire process without provoking any diagnostics, the resulting program can be run by giving its name to the Shell in response to the "%" prompt.

The next command you will need is *adb* (I). As a debugger, *adb* is better than average for assembly-language programs, marginally useful for C programs and virtually useless for Fortran.

Your programs can receive arguments from the command line just as system programs do. See *exec* (II).

Text processing. Almost all text is entered through the editor. The commands most often used to write text on a terminal are: *cat, pr,* and *nroff,* all in section I.

The *cat* command simply dumps ASCII text on the terminal, with no processing at all. The *pr* command paginates the text, supplies headings, and has a facility for multi-column output. *Nroff* is an elaborate text formatting program, and requires careful forethought in entering both the text and the formatting commands into the input file. *Nroff* produces output on a typewriter terminal.

Surprises. Certain commands provide inter-user communication. Even if you do not plan to use them, it would be well to learn something about them, because someone else may aim them at you.

To communicate with another user currently logged in, *write* (I) is used; *mail* (I) will leave a message whose presence will be announced to another user when he next logs in. The write-ups in the manual also suggest how to respond to the two commands if you are a target.

When you log in, a message-of-the-day may greet you before the first ‘‘%’’. The System Administrator, wishing to send a message-of-the-day, puts his data into /etc/motd, where it will be used as a general announcement each time someone logs on, until the message is erased from the file.

TABLE OF CONTENTS

I. COMMANDS

adb	debugger
admin	administer SCCS files
ar	archive and library maintainer
as	assembler
bas	basic
bc	arbitrary precision interactive language
cat	concatenate and print
cc	C compiler
chdir, cd	change working directory
chmod	change mode
chroot	change root directory for a command
cmp	compare two files
comm	print lines common to two files
cp	copy
cpall	copy all files to a directory
cref	make cross reference listing
date	print and set the date
dc	desk calculator
dd	convert and copy a file
delta	make an SCCS delta
diff	differential file comparator
diff3	3-way differential file comparison
dsw	delete interactively
du	summarize disk usage
echo	echo arguments
ed	text editor
eqn	typeset mathematics
exit	terminate command file
fc	Fortran compiler
file	determine format of file
find	find files
get	get generation from SCCS file
goto	command transfer
grep	search a file for a pattern
help	ask for help
if	conditional command
kill	terminate a process
X lc	LIL compiler
ld	link editor
X line	get terminal name

ln	make a link
login	sign onto UNIX
✗ lpr	line printer spooler
ls	list contents of directory
mail	send mail to designated users
mesg	permit or deny messages
mkdir	make a directory
mtm	magnetic tape manipulation
mv	move or rename a file
mvall	move all files to a directory
neqn	typeset mathematics on terminal
nice	run a command at low priority
nm	print name list
nohup	run a command immune to hangups
nroff	Format text for typewriter-like devices
od	octal dump
✗ onintr	specify interrupt processing for a command file
passwd	change login password
pfe	print floating exception
pr	print file
prof	display profile data
prt	print SCCS file
ps	process status
✗ pwd	working directory name
✗ read, open, onend	sequential file read
return	terminate profile or interrupt processing routine
rew	rewind tape
rm	remove (unlink) files
rmdir	remove directory
sed	stream editor
sh	shell (command interpreter)
shift	adjust Shell arguments
size	size of an object file
sleep	suspend execution for an interval
sort	sort or merge files
split	split a file into pieces
strip	remove symbols and relocation bits
stty	set typewriter options
sum	sum file
tail	deliver the last part of a file
tee	pipe fitting
time	time a command
tp	manipulate DECtape and magtape
tr	transliterate

troff	Format text for the Graphic Systems Phototypesetter
typo	find possible typos
uniq	report repeated lines in a file
wait	await completion of process
wall	write to all users
wc	word count
what	identify SCCS files
who	who is on the system
write	write to another user
yacc	yet another compiler-compiler

II. SYSTEM CALLS

intro	introduction to system calls
✗access	determine accessibility of file
✗acct	turn accounting on/off
alarm	activate alarm clock timer
break, brk, sbrk	change core allocation
call, lcall, vcall	create and execute a new process
chdir	change working directory
chmod	change mode of file
chown	change owner and group of a file
✗chroot	change root directory
close	close a file
creat	create a new file
csw	read console switches
dup	duplicate an open file descriptor
✗errlog	error logging
exec, execl, execv	execute a file
exit	terminate process
fork	spawn new process
fstat	get status of open file
getgid	get group identifications
getpid, (getppid) ✗	get process identification
getuid	get user identifications
gtty	get typewriter status
indir	indirect system call
✗ioctl	I/O control and parameter passing
kill	send signal to a process
link	link to a file
✗lock	semaphore operations
mknod	make a directory or a special file
mount	mount file system

X msg	send and receive messages
nice	set program priority
open	open for reading or writing
pause	suspend execution indefinitely
pipe	create an interprocess channel
profil	execution time profile
X ptrace	process trace
read	read from file
seek	move read/write pointer
setgid	set process group ID
setuid	set process user ID
signal	catch or ignore signals
sleep	stop execution for interval
stat	get file status
stime	set time
stty	set mode of typewriter
sync	update super-block
tell	get file offset
time	get date and time
times	get process times
umount	dismount file system
unlink	remove directory entry
wait	wait for process to terminate
write	write on a file

III. SUBROUTINES

abort	generate an IOT fault
abs, fabs	absolute value
alloc	core allocator
atan, atan2	arc tangent function
atof	convert ASCII to floating
atoi	convert ASCII to integer
compar	default comparison routine for qsort
crypt	password encoding
ctime, localtime, gmtime	convert date and time to ASCII
X dtol	floating point to double precision integer conversion
ecvt, fcvt	output conversion
end, etext, edata	last locations in program
exp	exponential function
floor, ceil	floor and ceiling functions
fmod	floating modulo function
fptrap	floating point interpreter
gamma	log gamma function

getarg, iargc	get command arguments from Fortran
getc, getw, fopen	buffered input
getchar	read character
✗ getpw	get name from UID
hmul	high-order product
hypot	calculate hypotenuse
ierror	catch Fortran errors
✗ itol	integer to long integer conversion
✗ ldiv, lrem	long division
✗ lnxx	return name of current terminal
✗ locv	long output conversion
log	natural logarithm
lseek	seek using a long offset
✗ ltod	double precision integer to floating point conversion
✗ ltoi	long integer to integer conversion
mesg	write message on typewriter
mktemp	make a unique named temporary file
monitor	prepare execution profile
nargs	argument count
nlist	get entries from name list
perror, sys_errlist, sys_nerr, errno	system error messages
pow	floating exponentiation
printf	formatted print
putc, putw, fcreat, fflush	buffered output
putchar, flush	write character
qsort	quicker sort
rand, srand	random number generator
reset, setexit	execute non-local goto
setfil	specify Fortran file name
sin, cos	trigonometric functions
sqrt	square root function

IV. DRIVERS

cm	Minitape interface
dc	DC-11 communications interface
dh	DH-11 communications multiplexer
dn	DN-11 ACU interface
dp	DP-11 201 data-phone interface
hd	half-duplex terminal interface
hp	RH-11/RP04 moving-head disk
hs	RH11/RS03-RS04 fixed-head disk file
ht	RH-11/TU-16 magtape interface
kl	KL-11 or DL-11 asynchronous interface

lp	line printer
mem, kmem, null	core memory
pc	PC-11 paper tape reader/punch
rf	RF11/RS11 fixed-head disk file
rk	RK-11/RK03 (or RK05) disk
rp	RP-11/RP03 moving-head disk
tc	TC-11/TU56 DECtape
tm	TM-11/TU-10 magtape interface
tty	interface to low speed asynchronous devices including typewriters

V. FILE FORMATS

a.out	assembler and link editor output
acct	Accounting file
ar	archive (library) file format
core	format of core image file
dir	format of directories
dump	incremental dump tape format
fs	format of file system volume
lines	script for the init process
passwd	password file
sccsfile	format of SCCS file
tp	DEC/mag tape formats
utmp	user information
wtmp	user login history

VI. USER PROGRAMS

agen	generate associative memory drivers
bj	the game of black jack
cal	print calendar
chess	the game of chess
cubic	three dimensional tic-tac-toe
factor	discover prime factors of a number
fed	edit form letter memory
form	form letter generator
gsi	interpret extended character set on GSI terminal
hyphen	find hyphenated words
lex	generate programs for simple lexical tasks
moo	guessing game
ptx	permuted index
reform	reformat text file
sno	Snobol interpreter
tbl	format tables for nroff or troff

tmac	macros for formatting manuscripts
ttt	the game of tic-tac-toe
wump	the game of hunt-the-wumpus

VII. TABLES

ascii	map of ASCII character set
greek	graphics for extended TTY-37 type-box
mtab	mounted file system table
tabs	set tab stops

VIII. SYSTEM PROGRAMS

ac	login accounting
accton	turn accounting on/off
boot procedures	UNIX startup
check	file system consistency check
chown	change owner
clri	clear i-node
crash	what to do when the system crashes
dcheck	file system directory consistency check
df	disk free
dump	incremental file system dump
errdemon	error logging daemon
errpt	process a report of logged errors
getty	set terminal modes, speed and line discipline
glob	generate command arguments
icheck	file system consistency check and interactive repair
init	process control initialization
ino	get the i-number of a file
iostat	report I/O statistics
load	load /unix
lpd	line printer daemon
mkconf	create configuration table and low core
mkfs	construct a file system
mknod	build special file
mkpt	make prototype file for use by mkfs
mount	mount file system
ncheck	generate names from i-numbers
reloc	relocate object files
restor	incremental file system restore
sa	Shell accounting
su	become privileged user
sync	update the super block

telinit	superuser communication with init
umount	dismount file system
update	periodically update the super block

PERMUTED INDEX

dp(IV) DP-11	201 data-phone interface
diff3(I)	3-way differential file comparison
dn(IV) DN-11	ACU interface
ascii(VII) map of	ASCII character set
atof(III) convert	ASCII to floating
atoi(III) convert	ASCII to integer
gmtime(III) convert date and time to	ASCII...ctime, localtime,
acct(V)	Accounting file
cc(I)	C compiler
dc(IV)	DC-11 communications interface
tp(V)	DEC/mag tape formats
tp(I) manipulate	DECtape and magtape
tc(IV) TC-11/TU56	DECtape
dh(IV)	DH-11 communications multiplexer
kl(IV) KL-11 or	DL-11 asynchronous interface
dn(IV)	DN-11 ACU interface
dp(IV)	DP-11 201 data-phone interface
troff(I)	Format text for the Graphic Systems Phototypesetter
nroff(I)	Format text for typewriter-like devices
fc(I)	Fortran compiler
ierror(III) catch	Fortran errors
setfil(III) specify	Fortran file name
iargc(III) get command arguments from	Fortran...getarg,
gsi(VI) interpret extended character set on	GSI terminal
troff(I) Format text for the	Graphic Systems Phototypesetter
setgid(II) set process group	ID
setuid(II) set process user	ID
ioctl(II)	I/O control and parameter passing
iostat(VIII) report	I/O statistics
abort(III) generate an	IOT fault
kl(IV)	KL-11 or DL-11 asynchronous interface
lc(I)	LIL compiler
cm(IV)	Minitape interface
pc(IV)	PC-11 paper tape reader/punch
Format text for the Graphic Systems	Phototypesetter...troff(I)
rf(IV)	RF11/RS11 fixed-head disk file
hp(IV)	RH-11/RP04 moving-head disk
hs(IV)	RH11/RS03-RS04 fixed-head disk file
ht(IV)	RH-11/TU-16 magtape interface
rk(IV) RK-11/RK03 (or	RK05) disk
rk(IV)	RK-11/RK03 (or RK05) disk
rp(IV)	RP-11/RP03 moving-head disk

delta(I) make an	SCCS delta
get(I) get generation from	SCCS file
prt(I) print	SCCS file
admin(I) administer	SCCS files
sccsfile(V) format of	SCCS file
what(I) identify	SCCS files
sa(VIII)	Shell accounting
shift(I) adjust	Shell arguments
sno(VI)	Snobol interpreter
troff(I) Format text for the Graphic	Systems Phototypesetter
tc(IV)	TC-11/TU56 DECtape
tm(IV)	TM-11/TU-10 magtape interface
greek(VII) graphics for extended	TTY-37 type-box
getpw(III) get name from	UID
boot procedures(VIII)	UNIX startup
login(I) sign onto	UNIX
	abort(III) generate an IOT fault
	abs, fabs(III) absolute value
abs, fabs(III)	absolute value
	ac(VIII) login accounting
	access(II) determine accessibility of file
access(II) determine	accessibility of file
acct(II) turn	accounting on/off
accton(VIII) turn	accounting on/off
ac(VIII) login	accounting
sa(VIII) Shell	accounting
	acct(II) turn accounting on/off
	acct(V) Accounting file
	accton(VIII) turn accounting on/off
alarm(II)	activate alarm clock timer
	adb(I) debugger
shift(I)	adjust Shell arguments
	admin(I) administer SCCS files
admin(I)	administer SCCS files
	agen(VI) generate associative memory drivers
alarm(II) activate	alarm clock timer
	alarm(II) activate alarm clock timer
	alloc(III) core allocator
break, brk, sbrk(II) change core	allocation
alloc(III) core	allocator
yacc(I) yet	another compiler-compiler
write(I) write to	another user
	a.out(V) assembler and link editor output
	ar(I) archive and library maintainer

	ar(V) archive (library) file format
bc(I)	arbitrary precision interactive language
atan, atan2(III)	arc tangent function
ar(I)	archive and library maintainer
ar(V)	archive (library) file format
nargs(III)	argument count
getarg, iargc(III) get command	arguments from Fortran
echo(I) echo	arguments
glob(VIII) generate command	arguments
shift(I) adjust Shell	arguments
	as(I) assembler
	ascii(VII) map of ASCII character set
help(I)	ask for help
a.out(V)	assembler and link editor output
as(I)	assembler
agen(VI) generate	associative memory drivers
tty(IV) interface to low speed	asynchronous devices including typewriters
kl(IV) KL-11 or DL-11	asynchronous interface
	atan, atan2(III) arc tangent function
atan,	atan2(III) arc tangent function
	atof(III) convert ASCII to floating
	atoi(III) convert ASCII to integer
wait(I)	await completion of process
	bas(I) basic
bas(I)	basic
	bc(I) arbitrary precision interactive language
su(VIII)	become privileged user
strip(I) remove symbols and relocation	bits
	bj(VI) the game of black jack
bj(VI) the game of	black jack
sync(VIII) update the super	block
update(VIII) periodically update the super	block
	boot procedures(VIII) UNIX startup
	break, brk, sbrk(II) change core allocation
break,	brk, sbrk(II) change core allocation
getc, getw, fopen(III)	buffered input
putc, putw, fcreat, fflush(III)	buffered output
mknod(VIII)	build special file
mkpt(VIII) make prototype file for use	by mkfs
	cal(VI) print calendar
hypot(III)	calculate hypotenuse
dc(I) desk	calculator
cal(VI) print	calendar
	call, lcall, vcall(II) create and execute a new process

indir(II) indirect system	call
intro(II) introduction to system	calls
	cat(I) concatenate and print
ierror(III)	catch Fortran errors
signal(II)	catch or ignore signals
	cc(I) C compiler
chdir,	cd(I) change working directory
floor,	ceil(III) floor and ceiling functions
floor, ceil(III) floor and	ceiling functions
break, brk, sbrk(II)	change core allocation
passwd(I)	change login password
chmod(II)	change mode of file
chmod(I)	change mode
chown(II)	change owner and group of a file
chown(VIII)	change owner
chroot(I)	change root directory for a command
chroot(II)	change root directory
chdir, cd(I)	change working directory
chdir(II)	change working directory
pipe(II) create an interprocess	channel
gsi(VI) interpret extended	character set on GSI terminal
ascii(VII) map of ASCII	character set
getchar(III) read	character
putchar, flush(III) write	character
	chdir, cd(I) change working directory
	chdir(II) change working directory
icheck(VIII) file system consistency	check and interactive repair
	check(VIII) file system consistency check
check(VIII) file system consistency	check
file system directory consistency	check...dcheck(VIII)
	chess(VI) the game of chess
chess(VI) the game of	chess
	chmod(I) change mode
	chmod(II) change mode of file
	chown(II) change owner and group of a file
	chown(VIII) change owner
	chroot(I) change root directory for a command
	chroot(II) change root directory
clri(VIII)	clear i-node
alarm(II) activate alarm	clock timer
close(II)	close a file
	close(II) close a file
	clri(VIII) clear i-node
	cm(IV) Minitape interface

	cmp(I) compare two files
	comm(I) print lines common to two files
getarg, iargc(III) get	command arguments from Fortran
glob(VIII) generate	command arguments
nice(I) run a	command at low priority
exit(I) terminate	command file
specify interrupt processing for a	command file...onintr(I)
nohup(I) run a	command immune to hangups
sh(I) shell	(command interpreter)
goto(I)	command transfer
chroot(I) change root directory for a	command
if(I) conditional	command
time(I) time a	command
comm(I) print lines	common to two files
telinit(VIII) superuser	communication with init
dc(IV) DC-11	communications interface
dh(IV) DH-11	communications multiplexer
	compar(III) default comparison routine for qsort
diff(I) differential file	comparator
cmp(I)	compare two files
compar(III) default	comparison routine for qsort
diff3(I) 3-way differential file	comparison
cc(I) C	compiler
yacc(I) yet another	compiler-compiler
fc(I) Fortran	compiler
lc(I) LIL	compiler
wait(I) await	completion of process
cat(I)	concatenate and print
if(I)	conditional command
mkconf(VIII) create	configuration table and low core
icheck(VIII) file system	consistency check and interactive repair
check(VIII) file system	consistency check
dcheck(VIII) file system directory	consistency check
csw(II) read	console switches
mkfs(VIII)	construct a file system
ls(I) list	contents of directory
ioctl(II) I/O	control and parameter passing
init(VIII) process	control initialization
floating point to double precision integer	conversion...dtol(III)
ecvt, fcvt(III) output	conversion
itol(III) integer to long integer	conversion
locv(III) long output	conversion
double precision integer to floating point	conversion...ltod(III)
ltoi(III) long integer to integer	conversion

atof(III)	convert ASCII to floating
atoi(III)	convert ASCII to integer
dd(I)	convert and copy a file
ctime, localtime, gmtime(III)	convert date and time to ASCII
dd(I) convert and	copy a file
cpall(I)	copy all files to a directory
cp(I)	copy
break, brk, sbrk(II) change	core allocation
alloc(III)	core allocator
core(V) format of	core image file
mem, kmem, null(IV)	core memory
	core(V) format of core image file
create configuration table and low	core...mkconf(VIII)
sin,	cos(III) trigonometric functions
nargs(III) argument	count
wc(I) word	count
	cp(I) copy
	cpall(I) copy all files to a directory
	crash(VIII) what to do when the system crashes
crash(VIII) what to do when the system	crashes
	creat(II) create a new file
creat(II)	create a new file
pipe(II)	create an interprocess channel
call, lcall, vcall(II)	create and execute a new process
mkconf(VIII)	create configuration table and low core
	cref(I) make cross reference listing
cref(I) make	cross reference listing
	crypt(III) password encoding
	csw(II) read console switches
ASCII...	ctime, localtime, gmtime(III) convert date and time to
	cubic(VI) three dimensional tic-tac-toe
lnxx(III) return name of	current terminal
errdemon(VIII) error logging	daemon
lpd(VIII) line printer	daemon
dp(IV) DP-11 201	data-phone interface
prof(I) display profile	data
ctime, localtime, gmtime(III) convert	date and time to ASCII
time(II) get	date and time
	date(I) print and set the date
date(I) print and set the	date
	dc(I) desk calculator
	dc(IV) DC-11 communications interface
	dcheck(VIII) file system directory consistency check
	dd(I) convert and copy a file

adb(I)	debugger
compar(III)	default comparison routine for qsort
dsw(I)	delete interactively
tail(I)	deliver the last part of a file
	delta(I) make an SCCS delta
delta(I) make an SCCS	delta
mesg(I) permit or	deny messages
dup(II) duplicate an open file	descriptor
mail(I) send mail to	designated users
dc(I)	desk calculator
access(II)	determine accessibility of file
file(I)	determine format of file
tty(IV) interface to low speed asynchronous	devices including typewriters
nroff(I) Format text for typewriter-like	devices
	df(VIII) disk free
	dh(IV) DH-11 communications multiplexer
	diff3(I) 3-way differential file comparison
	diff(I) differential file comparator
diff(I)	differential file comparator
diff3(I) 3-way	differential file comparison
cubic(VI) three	dimensional tic-tac-toe
	dir(V) format of directories
dir(V) format of	directories
dcheck(VIII) file system	directory consistency check
unlink(II) remove	directory entry
chroot(I) change root	directory for a command
pwd(I) working	directory name
mknod(II) make a	directory or a special file
chdir, cd(I) change working	directory
chdir(II) change working	directory
chroot(II) change root	directory
cpall(I) copy all files to a	directory
ls(I) list contents of	directory
mkdir(I) make a	directory
mvall(I) move all files to a	directory
rmdir(I) remove	directory
set terminal modes, speed and line	discipline...getty(VIII)
factor(VI)	discover prime factors of a number
hs(IV) RH11/RS03-RS04 fixed-head	disk file
rf(IV) RF11/RS11 fixed-head	disk file
df(VIII)	disk free
du(I) summarize	disk usage
hp(IV) RH-11/RP04 moving-head	disk
rk(IV) RK-11/RK03 (or RK05)	disk

rp(IV) RP-11/RP03 moving-head	disk
umount(II)	dismount file system
umount(VIII)	dismount file system
prof(I)	display profile data
ldiv, lrem(III) long	division
	dn(IV) DN-11 ACU interface
dtol(III) floating point to	double precision integer conversion
ltod(III)	double precision integer to floating point conversion
	dp(IV) DP-11 201 data-phone interface
agen(VI) generate associative memory	drivers
	dsw(I) delete interactively
conversion...	dtol(III) floating point to double precision integer
	du(I) summarize disk usage
dump(V) incremental	dump tape format
	dump(V) incremental dump tape format
	dump(VIII) incremental file system dump
dump(VIII) incremental file system	dump
od(I) octal	dump
	dup(II) duplicate an open file descriptor
dup(II)	duplicate an open file descriptor
echo(I)	echo arguments
	echo(I) echo arguments
	ecvt, fcvt(III) output conversion
	ed(I) text editor
end, etext,	edata(III) last locations in program
fed(VI)	edit form letter memory
a.out(V) assembler and link	editor output
ed(I) text	editor
ld(I) link	editor
sed(I) stream	editor
crypt(III) password	encoding
	end, etext, edata(III) last locations in program
nlist(III) get	entries from name list
unlink(II) remove directory	entry
	eqn(I) typeset mathematics
	errdemon(VIII) error logging daemon
	errlog(II) error logging
perror, sys_errlist, sys_nerr,	errno(III) system error messages
errdemon(VIII)	error logging daemon
errlog(II)	error logging
sys_nerr, errno(III) system	error messages...perror, sys_errlist,
errpt(VIII) process a report of logged	errors
ierror(III) catch Fortran	errors
	errpt(VIII) process a report of logged errors

end,	etext, edata(III) last locations in program
pfe(I) print floating	exception
	exec, execl, execv(II) execute a file
exec,	execl, execv(II) execute a file
exec, execl, execv(II)	execute a file
call, lcall, vcall(II) create and	execute a new process
reset, setexit(III)	execute non-local goto
sleep(I) suspend	execution for an interval
sleep(II) stop	execution for interval
pause(II) suspend	execution indefinitely
monitor(III) prepare	execution profile
profil(II)	execution time profile
exec, execl,	execv(II) execute a file
	exit(I) terminate command file
	exit(II) terminate process
	exp(III) exponential function
exp(III)	exponential function
pow(III) floating	exponentiation
greek(VII) graphics for	extended TTY-37 type-box
gsi(VI) interpret	extended character set on GSI terminal
abs,	fabs(III) absolute value
	factor(VI) discover prime factors of a number
factor(VI) discover prime	factors of a number
abort(III) generate an IOT	fault
	fc(I) Fortran compiler
putc, putw,	fcreat, fflush(III) buffered output
ecvt,	fcvt(III) output conversion
	fed(VI) edit form letter memory
putc, putw, fcreat,	fflush(III) buffered output
diff(I) differential	file comparator
diff3(I) 3-way differential	file comparison
dup(II) duplicate an open	file descriptor
grep(I) search a	file for a pattern
mkpt(VIII) make prototype	file for use by mkfs
ar(V) archive (library)	file format
split(I) split a	file into pieces
setfil(III) specify Fortran	file name
tell(II) get	file offset
read, open, onend(I) sequential	file read
stat(II) get	file status
icheck(VIII)	file system consistency check and interactive repair
check(VIII)	file system consistency check
dcheck(VIII)	file system directory consistency check
dump(VIII) incremental	file system dump

restor(VIII) incremental	file system restore
mtab(VII) mounted	file system table
fs(V) format of	file system volume
mkfs(VIII) construct a	file system
mount(II) mount	file system
mount(VIII) mount	file system
umount(II) dismount	file system
umount(VIII) dismount	file system
	file(I) determine format of file
access(II) determine accessibility of	file
acct(V) Accounting	file
chmod(II) change mode of	file
chown(II) change owner and group of a	file
close(II) close a	file
core(V) format of core image	file
creat(II) create a new	file
dd(I) convert and copy a	file
exec, execl, execv(II) execute a	file
exit(I) terminate command	file
file(I) determine format of	file
fstat(II) get status of open	file
get(I) get generation from SCCS	file
hs(IV) RH11/RS03-RS04 fixed-head disk	file
ino(VIII) get the i-number of a	file
link(II) link to a	file
mknod(II) make a directory or a special	file
mknod(VIII) build special	file
mktemp(III) make a unique named temporary	file
mv(I) move or rename a	file
specify interrupt processing for a command	file...onintr(I)
passwd(V) password	file
pr(I) print	file
prt(I) print SCCS	file
read(II) read from	file
reform(VI) reformat text	file
rf(IV) RF11/RS11 fixed-head disk	file
cpall(I) copy all	files to a directory
mvall(I) move all	files to a directory
admin(I) administer SCCS	files
sccsfile(V) format of SCCS	file
cmp(I) compare two	files
comm(I) print lines common to two	files
find(I) find	files
size(I) size of an object	file

reloc(VIII) relocate object	files
rm(I) remove (unlink)	files
sort(I) sort or merge	files
sum(I) sum	file
what(I) identify SCCS	files
tail(I) deliver the last part of a	file
uniq(I) report repeated lines in a	file
write(II) write on a	file
find(I)	find files
hyphen(VI)	find hyphenated words
typo(I)	find possible typos
	find(I) find files
tee(I) pipe	fitting
hs(IV) RH11/RS03-RS04	fixed-head disk file
rf(IV) RF11/RS11	fixed-head disk file
pfe(I) print	floating exception
pow(III)	floating exponentiation
fmod(III)	floating modulo function
ltod(III) double precision integer to	floating point conversion
fptrap(III)	floating point interpreter
dtol(III)	floating point to double precision integer conversion
atof(III) convert ASCII to	floating
floor, ceil(III)	floor and ceiling functions
	floor, ceil(III) floor and ceiling functions
putchar,	flush(III) write character
	fmod(III) floating modulo function
getc, getw,	fopen(III) buffered input
	fork(II) spawn new process
form(VI)	form letter generator
fed(VI) edit	form letter memory
	form(VI) form letter generator
sccsfile(V)	format of SCCS file
core(V)	format of core image file
dir(V)	format of directories
fs(V)	format of file system volume
file(I) determine	format of file
tbl(VI)	format tables for nroff or troff
ar(V) archive (library) file	format
dump(V) incremental dump tape	format
tp(V) DEC/mag tape	formats
printf(III)	formatted print
tmac(VI) macros for	formatting manuscripts
	fptrap(III) floating point interpreter
df(VIII) disk	free

getarg, iargc(III) get command arguments	from Fortran
get(I) get generation	from SCCS file
getpw(III) get name	from UID
read(II) read	from file
ncheck(VIII) generate names	from i-numbers
nlist(III) get entries	from name list
	fs(V) format of file system volume
	fstat(II) get status of open file
atan, atan2(III) arc tangent	function
exp(III) exponential	function
fmod(III) floating modulo	function
gamma(III) log gamma	function
floor, ceil(III) floor and ceiling	functions
sqrt(III) square root	function
sin, cos(III) trigonometric	functions
bj(VI) the	game of black jack
chess(VI) the	game of chess
wump(VI) the	game of hunt-the-wumpus
ttt(VI) the	game of tic-tac-toe
moo(VI) guessing	game
gamma(III) log	gamma function
	gamma(III) log gamma function
abort(III)	generate an IOT fault
agen(VI)	generate associative memory drivers
glob(VIII)	generate command arguments
ncheck(VIII)	generate names from i-numbers
lex(VI)	generate programs for simple lexical tasks
get(I) get	generation from SCCS file
form(VI) form letter	generator
rand, srand(III) random number	generator
getarg, iargc(III)	get command arguments from Fortran
time(II)	get date and time
nlist(III)	get entries from name list
tell(II)	get file offset
stat(II)	get file status
get(I)	get generation from SCCS file
getgid(II)	get group identifications
getpw(III)	get name from UID
getpid, getppid(II)	get process identification
times(II)	get process times
fstat(II)	get status of open file
line(I)	get terminal name
ino(VIII)	get the i-number of a file

gtty(II)	get typewriter status
getuid(II)	get user identifications
	get(I) get generation from SCCS file
	getarg, iargc(III) get command arguments from Fortran
	getc, getw, fopen(III) buffered input
	getchar(III) read character
	getgid(II) get group identifications
	getpid, getppid(II) get process identification
getpid,	getppid(II) get process identification
	getpw(III) get name from UID
discipline...	getty(VIII) set terminal modes, speed and line
	getuid(II) get user identifications
getc,	getw, fopen(III) buffered input
	glob(VIII) generate command arguments
ctime, localtime,	gmtime(III) convert date and time to ASCII
	goto(I) command transfer
reset, setexit(III) execute non-local	goto
greek(VII)	graphics for extended TTY-37 type-box
	greek(VII) graphics for extended TTY-37 type-box
	grep(I) search a file for a pattern
setgid(II) set process	group ID
getgid(II) get	group identifications
chown(II) change owner and	group of a file
	gsi(VI) interpret extended character set on GSI terminal
	gtty(II) get typewriter status
moo(VI)	guessing game
hd(IV)	half-duplex terminal interface
nohup(I) run a command immune to	hangups
	hd(IV) half-duplex terminal interface
	help(I) ask for help
help(I) ask for	help
hmul(III)	high-order product
wtmp(V) user login	history
	hmul(III) high-order product
	hp(IV) RH-11/RP04 moving-head disk
	hs(IV) RH11/RS03-RS04 fixed-head disk file
	ht(IV) RH-11/TU-16 magtape interface
wump(VI) the game of	hunt-the-wumpus
	hyphen(VI) find hyphenated words
hyphen(VI) find	hyphenated words
	hypot(III) calculate hypotenuse
hypot(III) calculate	hypotenuse
getarg,	iargc(III) get command arguments from Fortran
interactive repair...	icheck(VIII) file system consistency check and

getpid, getppid(II) get process	identification
getgid(II) get group	identifications
getuid(II) get user	identifications
what(I)	identify SCCS files
	ierror(III) catch Fortran errors
	if(I) conditional command
signal(II) catch or	ignore signals
core(V) format of core	image file
nohup(I) run a command	immune to hangups
interface to low speed asynchronous devices	including typewriters...tty(IV)
dump(V)	incremental dump tape format
dump(VIII)	incremental file system dump
restor(VIII)	incremental file system restore
pause(II) suspend execution	indefinitely
ptx(VI) permuted	index
	indir(II) indirect system call
indir(II)	indirect system call
utmp(V) user	information
lines(V) script for the	init process
	init(VIII) process control initialization
init(VIII) process control	initialization
telinit(VIII) superuser communication with	init
	ino(VIII) get the i-number of a file
clri(VIII) clear	i-node
getc, getw, fopen(III) buffered	input
floating point to double precision	integer conversion...dtol(III)
itol(III) integer to long	integer conversion
ltoi(III) long integer to	integer conversion
ltod(III) double precision	integer to floating point conversion
ltoi(III) long	integer to integer conversion
itol(III)	integer to long integer conversion
atoi(III) convert ASCII to	integer
bc(I) arbitrary precision	interactive language
file system consistency check and	interactive repair...icheck(VIII)
dsw(I) delete	interactively
typewriters...tty(IV)	interface to low speed asynchronous devices including
cm(IV) Minitape	interface
dc(IV) DC-11 communications	interface
dn(IV) DN-11 ACU	interface
dp(IV) DP-11 201 data-phone	interface
hd(IV) half-duplex terminal	interface
ht(IV) RH-11/TU-16 magtape	interface
kl(IV) KL-11 or DL-11 asynchronous	interface
tm(IV) TM-11/TU-10 magtape	interface

gsi(VI)	interpret extended character set on GSI terminal
fptrap(III) floating point	interpreter
sh(I) shell (command	interpreter)
sno(VI) Snobol	interpreter
pipe(II) create an	interprocess channel
onintr(I) specify	interrupt processing for a command file
return(I) terminate profile or	interrupt processing routine
sleep(I) suspend execution for an	interval
sleep(II) stop execution for	interval
	intro(II) introduction to system calls
intro(II)	introduction to system calls
ino(VIII) get the	i-number of a file
ncheck(VIII) generate names from	i-numbers
	ioctl(II) I/O control and parameter passing
	iostat(VIII) report I/O statistics
	itol(III) integer to long integer conversion
bj(VI) the game of black	jack
	kill(I) terminate a process
	kill(II) send signal to a process
	kl(IV) KL-11 or DL-11 asynchronous interface
mem,	kmem, null(IV) core memory
bc(I) arbitrary precision interactive	language
end, etext, edata(III)	last locations in program
tail(I) deliver the	last part of a file
	lc(I) LIL compiler
call,	lcall, vcall(II) create and execute a new process
	ld(I) link editor
	ldiv, lrem(III) long division
form(VI) form	letter generator
fed(VI) edit form	letter memory
	lex(VI) generate programs for simple lexical tasks
lex(VI) generate programs for simple	lexical tasks
ar(V) archive	(library) file format
ar(I) archive and	library maintainer
getty(VIII) set terminal modes, speed and	line discipline
lpd(VIII)	line printer daemon
lpr(I)	line printer spooler
lp(IV)	line printer
	line(I) get terminal name
comm(I) print	lines common to two files
uniq(I) report repeated	lines in a file
	lines(V) script for the init process
a.out(V) assembler and	link editor output
ld(I)	link editor

link(II)	link to a file
	link(II) link to a file
ln(I) make a	link
ls(I)	list contents of directory
cref(I) make cross reference	listing
nlist(III) get entries from name	list
nm(I) print name	list
	ln(I) make a link
	lnxx(III) return name of current terminal
load(VIII)	load /unix
	load(VIII) load /unix
ctime,	localtime, gmtime(III) convert date and time to ASCII
end, etext, edata(III) last	locations in program
	lock(II) semaphore operations
	locv(III) long output conversion
gamma(III)	log gamma function
	log(III) natural logarithm
log(III) natural	logarithm
errpt(VIII) process a report of	logged errors
errdemon(VIII) error	logging daemon
errlog(II) error	logging
ac(VIII)	login accounting
wtmp(V) user	login history
passwd(I) change	login password
	login(I) sign onto UNIX
ldiv, lrem(III)	long division
itol(III) integer to	long integer conversion
ltoi(III)	long integer to integer conversion
lseek(III) seek using a	long offset
locv(III)	long output conversion
mkconf(VIII) create configuration table and	low core
nice(I) run a command at	low priority
tty(IV) interface to	low speed asynchronous devices including typewriters
	lp(IV) line printer
	lpd(VIII) line printer daemon
	lpr(I) line printer spooler
ldiv,	lrem(III) long division
	ls(I) list contents of directory
	lseek(III) seek using a long offset
conversion...	ltod(III) double precision integer to floating point
	ltoi(III) long integer to integer conversion
tmac(VI)	macros for formatting manuscripts
mtm(I)	magnetic tape manipulation
ht(IV) RH-11/TU-16	magtape interface

tm(IV) TM-11/TU-10	magtape interface
tp(I) manipulate DECtape and	magtape
mail(I) send	mail to designated users
	mail(I) send mail to designated users
ar(I) archive and library	maintainer
mknod(II)	make a directory or a special file
mkdir(I)	make a directory
ln(I)	make a link
mktemp(III)	make a unique named temporary file
delta(I)	make an SCCS delta
cref(I)	make cross reference listing
mkpt(VIII)	make prototype file for use by mkfs
tp(I)	manipulate DECtape and magtape
mtm(I) magnetic tape	manipulation
tmac(VI) macros for formatting	manuscripts
ascii(VII)	map of ASCII character set
neqn(I) typeset	mathematics on terminal
eqn(I) typeset	mathematics
	mem, kmem, null(IV) core memory
agen(VI) generate associative	memory drivers
fed(VI) edit form letter	memory
mem, kmem, null(IV) core	memory
sort(I) sort or	merge files
	mesg(I) permit or deny messages
	mesg(III) write message on typewriter
mesg(III) write	message on typewriter
mesg(I) permit or deny	messages
msg(II) send and receive	messages
sys_nerr, errno(III) system error	messages...perror, sys_errlist,
	mkconf(VIII) create configuration table and low core
	mkdir(I) make a directory
	mkfs(VIII) construct a file system
mkpt(VIII) make prototype file for use by	mkfs
	mknod(II) make a directory or a special file
	mknod(VIII) build special file
	mkpt(VIII) make prototype file for use by mkfs
	mktemp(III) make a unique named temporary file
chmod(II) change	mode of file
stty(II) set	mode of typewriter
chmod(I) change	mode
getty(VIII) set terminal	modes, speed and line discipline
fmod(III) floating	modulo function
	monitor(III) prepare execution profile
	moo(VI) guessing game

mount(II)	mount file system
mount(VIII)	mount file system
	mount(II) mount file system
	mount(VIII) mount file system
mtab(VII)	mounted file system table
mvall(I)	move all files to a directory
mv(I)	move or rename a file
seek(II)	move read/write pointer
hp(IV) RH-11/RP04	moving-head disk
rp(IV) RP-11/RP03	moving-head disk
	msg(II) send and receive messages
	mtab(VII) mounted file system table
	mtm(I) magnetic tape manipulation
dh(IV) DH-11 communications	multiplexer
	mv(I) move or rename a file
	mvall(I) move all files to a directory
getpw(III) get	name from UID
nlist(III) get entries from	name list
nm(I) print	name list
lnxx(III) return	name of current terminal
mktemp(III) make a unique	named temporary file
line(I) get terminal	name
pwd(I) working directory	name
ncheck(VIII) generate	names from i-numbers
setfil(III) specify Fortran file	name
	nargs(III) argument count
log(III)	natural logarithm
	ncheck(VIII) generate names from i-numbers
	neqn(I) typeset mathematics on terminal
creat(II) create a	new file
call, lcall, vcall(II) create and execute a	new process
fork(II) spawn	new process
	nice(I) run a command at low priority
	nice(II) set program priority
	nlist(III) get entries from name list
	nm(I) print name list
	nohup(I) run a command immune to hangups
reset, setexit(III) execute	non-local goto
tbl(VI) format tables for	nroff or troff
	nroff(I) Format text for typewriter-like devices
mem, kmem,	null(IV) core memory
rand, srand(III) random	number generator
factor(VI) discover prime factors of a	number

size(I) size of an	object file
reloc(VIII) relocate	object files
od(I)	octal dump
	od(I) octal dump
lseek(III) seek using a long	offset
tell(II) get file	offset
read, open,	onend(I) sequential file read
file...	onintr(I) specify interrupt processing for a command
acct(II) turn accounting	on/off
accton(VIII) turn accounting	on/off
login(I) sign	onto UNIX
dup(II) duplicate an	open file descriptor
fstat(II) get status of	open file
open(II)	open for reading or writing
read,	open, onend(I) sequential file read
	open(II) open for reading or writing
lock(II) semaphore	operations
stty(I) set typewriter	options
rk(IV) RK-11/RK03	(or RK05) disk
ecvt, fcvt(III)	output conversion
locv(III) long	output conversion
a.out(V) assembler and link editor	output
putc, putw, fcreat, fflush(III) buffered	output
chown(II) change	owner and group of a file
chown(VIII) change	owner
pc(IV) PC-11	paper tape reader/punch
ioctl(II) I/O control and	parameter passing
tail(I) deliver the last	part of a file
ioctl(II) I/O control and parameter	passing
	passwd(I) change login password
	passwd(V) password file
crypt(III)	password encoding
passwd(V)	password file
passwd(I) change login	password
grep(I) search a file for a	pattern
	pause(II) suspend execution indefinitely
	pc(IV) PC-11 paper tape reader/punch
update(VIII)	periodically update the super block
mesg(I)	permit or deny messages
ptx(VI)	permuted index
error messages...	perror, sys_errlist, sys_nerr, errno(III) system
	pfe(I) print floating exception
split(I) split a file into	pieces
tee(I)	pipe fitting

	pipe(II) create an interprocess channel
double precision integer to floating	point conversion...ltod(III)
fptrap(III) floating	point interpreter
dtol(III) floating	point to double precision integer conversion
seek(II) move read/write	pointer
typo(I) find	possible typos
	pow(III) floating exponentiation
	pr(I) print file
dtol(III) floating point to double	precision integer conversion
ltod(III) double	precision integer to floating point conversion
bc(I) arbitrary	precision interactive language
monitor(III)	prepare execution profile
factor(VI) discover	prime factors of a number
prt(I)	print SCCS file
date(I)	print and set the date
cal(VI)	print calendar
pr(I)	print file
pfe(I)	print floating exception
comm(I)	print lines common to two files
nm(I)	print name list
cat(I) concatenate and	print
lpd(VIII) line	printer daemon
lpr(I) line	printer spooler
lp(IV) line	printer
	printf(III) formatted print
printf(III) formatted	print
nice(I) run a command at low	priority
nice(II) set program	priority
su(VIII) become	privileged user
boot	procedures(VIII) UNIX startup
errpt(VIII)	process a report of logged errors
init(VIII)	process control initialization
setgid(II) set	process group ID
getpid, getppid(II) get	process identification
ps(I)	process status
times(II) get	process times
wait(II) wait for	process to terminate
ptrace(II)	process trace
setuid(II) set	process user ID
lcall, vcall(II) create and execute a new	process...call,
exit(II) terminate	process
fork(II) spawn new	process
onintr(I) specify interrupt	processing for a command file
return(I) terminate profile or interrupt	processing routine

kill(I) terminate a	process
kill(II) send signal to a	process
lines(V) script for the init	process
wait(I) await completion of	process
hmul(III) high-order	product
	prof(I) display profile data
	profil(II) execution time profile
prof(I) display	profile data
return(I) terminate	profile or interrupt processing routine
monitor(III) prepare execution	profile
profil(II) execution time	profile
nice(II) set	program priority
end, etext, edata(III) last locations in	program
lex(VI) generate	programs for simple lexical tasks
mkpt(VIII) make	prototype file for use by mkfs
	prt(I) print SCCS file
	ps(I) process status
	ptrace(II) process trace
	ptx(VI) permuted index
	putc, putw, fcreat, fflush(III) buffered output
	putchar, flush(III) write character
putc,	putw, fcreat, fflush(III) buffered output
	pwd(I) working directory name
	qsort(III) quicker sort
compar(III) default comparison routine for	qsort
qsort(III)	quicker sort
	rand, srand(III) random number generator
rand, srand(III)	random number generator
getchar(III)	read character
csw(II)	read console switches
read(II)	read from file
	read, open, onend(I) sequential file read
	read(II) read from file
pc(IV) PC-11 paper tape	reader/punch
open(II) open for	reading or writing
read, open, onend(I) sequential file	read
seek(II) move	read/write pointer
msg(II) send and	receive messages
cref(I) make cross	reference listing
	reform(VI) reformat text file
reform(VI)	reformat text file
	reloc(VIII) relocate object files
reloc(VIII)	relocate object files
strip(I) remove symbols and	relocation bits

unlink(II)	remove directory entry
rmdir(I)	remove directory
strip(I)	remove symbols and relocation bits
rm(I)	remove (unlink) files
mv(I) move or	rename a file
system consistency check and interactive	repair...icheck(VIII) file
uniq(I) report	repeated lines in a file
iostat(VIII)	report I/O statistics
errpt(VIII) process a	report of logged errors
uniq(I)	report repeated lines in a file
	reset, setexit(III) execute non-local goto
	restor(VIII) incremental file system restore
restor(VIII) incremental file system	restore
lnxx(III)	return name of current terminal
routine...	return(I) terminate profile or interrupt processing
	rew(I) rewind tape
rew(I)	rewind tape
	rf(IV) RF11/RS11 fixed-head disk file
	rk(IV) RK-11/RK03 (or RK05) disk
	rm(I) remove (unlink) files
	rmdir(I) remove directory
chroot(I) change	root directory for a command
chroot(II) change	root directory
sqrt(III) square	root function
compar(III) default comparison	routine for qsort
terminate profile or interrupt processing	routine...return(I)
	rp(IV) RP-11/RP03 moving-head disk
nice(I)	run a command at low priority
nohup(I)	run a command immune to hangups
	sa(VIII) Shell accounting
break, brk,	sbrk(II) change core allocation
	sccsfile(V) format of SCCS file
lines(V)	script for the init process
grep(I)	search a file for a pattern
	sed(I) stream editor
lseek(III)	seek using a long offset
	seek(II) move read/write pointer
lock(II)	semaphore operations
msg(II)	send and receive messages
mail(I)	send mail to designated users
kill(II)	send signal to a process
read, open, onend(I)	sequential file read
stty(II)	set mode of typewriter
gsi(VI) interpret extended character	set on GSI terminal

setgid(II)	set process group ID
setuid(II)	set process user ID
nice(II)	set program priority
tabs(VII)	set tab stops
getty(VIII)	set terminal modes, speed and line discipline
date(I) print and	set the date
stime(II)	set time
stty(I)	set typewriter options
ascii(VII) map of ASCII character	set
reset,	setexit(III) execute non-local goto
	setfil(III) specify Fortran file name
	setgid(II) set process group ID
	setuid(II) set process user ID
	sh(I) shell (command interpreter)
sh(I)	shell (command interpreter)
	shift(I) adjust Shell arguments
login(I)	sign onto UNIX
kill(II) send	signal to a process
	signal(II) catch or ignore signals
signal(II) catch or ignore	signals
lex(VI) generate programs for	simple lexical tasks
	sin, cos(III) trigonometric functions
size(I)	size of an object file
	size(I) size of an object file
	sleep(I) suspend execution for an interval
	sleep(II) stop execution for interval
	sno(VI) Snobol interpreter
sort(I)	sort or merge files
	sort(I) sort or merge files
qsort(III) quicker	sort
fork(II)	spawn new process
mknod(II) make a directory or a	special file
mknod(VIII) build	special file
setfil(III)	specify Fortran file name
onintr(I)	specify interrupt processing for a command file
getty(VIII) set terminal modes,	speed and line discipline
tty(IV) interface to low	speed asynchronous devices including typewriters
split(I)	split a file into pieces
	split(I) split a file into pieces
lpr(I) line printer	spooler
	sqrt(III) square root function
sqrt(III)	square root function
rand,	srand(III) random number generator
boot procedures(VIII) UNIX	startup

	stat(II) get file status
iostat(VIII) report I/O	statistics
fstat(II) get	status of open file
gtty(II) get typewriter	status
ps(I) process	status
stat(II) get file	status
	stime(II) set time
sleep(II)	stop execution for interval
tabs(VII) set tab	stops
sed(I)	stream editor
	strip(I) remove symbols and relocation bits
	stty(I) set typewriter options
	stty(II) set mode of typewriter
	su(VIII) become privileged user
sum(I)	sum file
	sum(I) sum file
du(I)	summarize disk usage
sync(VIII) update the	super block
update(VIII) periodically update the	super block
sync(II) update	super-block
telinit(VIII)	superuser communication with init
sleep(I)	suspend execution for an interval
pause(II)	suspend execution indefinitely
csw(II) read console	switches
strip(I) remove	symbols and relocation bits
	sync(II) update super-block
	sync(VIII) update the super block
messages...perror,	sys_errlist, sys_nerr, errno(III) system error
perror, sys_errlist,	sys_nerr, errno(III) system error messages
indir(II) indirect	system call
intro(II) introduction to	system calls
icheck(VIII) file	system consistency check and interactive repair
check(VIII) file	system consistency check
crash(VIII) what to do when the	system crashes
dcheck(VIII) file	system directory consistency check
dump(VIII) incremental file	system dump
sys_errlist, sys_nerr, errno(III)	system error messages...perror,
restor(VIII) incremental file	system restore
mtab(VII) mounted file	system table
fs(V) format of file	system volume
mkfs(VIII) construct a file	system
mount(II) mount file	system
mount(VIII) mount file	system
umount(II) dismount file	system

umount(VIII) dismount file	system
who(I) who is on the	system
tabs(VII) set	tab stops
mkconf(VIII) create configuration	table and low core
mtab(VII) mounted file system	table
tbl(VI) format	tables for nroff or troff
	tabs(VII) set tab stops
	tail(I) deliver the last part of a file
atan, atan2(III) arc	tangent function
dump(V) incremental dump	tape format
tp(V) DEC/mag	tape formats
mtm(I) magnetic	tape manipulation
pc(IV) PC-11 paper	tape reader/punch
rew(I) rewind	tape
generate programs for simple lexical	tasks...lex(VI)
	tbl(VI) format tables for nroff or troff
	tc(IV) TC-11/TU56 DECtape
	tee(I) pipe fitting
	telinit(VIII) superuser communication with init
	tell(II) get file offset
mktemp(III) make a unique named	temporary file
hd(IV) half-duplex	terminal interface
getty(VIII) set	terminal modes, speed and line discipline
line(I) get	terminal name
interpret extended character set on GSI	terminal...gsi(VI)
lnxx(III) return name of current	terminal
neqn(I) typeset mathematics on	terminal
kill(I)	terminate a process
exit(I)	terminate command file
exit(II)	terminate process
return(I)	terminate profile or interrupt processing routine
wait(II) wait for process to	terminate
ed(I)	text editor
reform(VI) reformat	text file
troff(I) Format	text for the Graphic Systems Phototypesetter
nroff(I) Format	text for typewriter-like devices
cubic(VI)	three dimensional tic-tac-toe
cubic(VI) three dimensional	tic-tac-toe
ttt(VI) the game of	tic-tac-toe
time(I)	time a command
profil(II) execution	time profile
localtime, gmtime(III) convert date and	time to ASCII...ctime,
	time(I) time a command
	time(II) get date and time

alarm(II) activate alarm clock	timer
	times(II) get process times
stime(II) set	time
times(II) get process	times
time(II) get date and	time
	tm(IV) TM-11/TU-10 magtape interface
	tmac(VI) macros for formatting manuscripts
	tp(I) manipulate DECtape and magtape
	tp(V) DEC/mag tape formats
	tr(I) transliterate
ptrace(II) process	trace
goto(I) command	transfer
tr(I)	transliterate
sin, cos(III)	trigonometric functions
Phototypesetter...	troff(I) Format text for the Graphic Systems
tbl(VI) format tables for nroff or	troff
	ttt(VI) the game of tic-tac-toe
including typewriters...	tty(IV) interface to low speed asynchronous devices
acct(II)	turn accounting on/off
accton(VIII)	turn accounting on/off
cmp(I) compare	two files
comm(I) print lines common to	two files
greek(VII) graphics for extended TTY-37	type-box
neqn(I)	typeset mathematics on terminal
eqn(I)	typeset mathematics
stty(I) set	typewriter options
gtty(II) get	typewriter status
nroff(I) Format text for	typewriter-like devices
mesg(III) write message on	typewriter
stty(II) set mode of	typewriter
to low speed asynchronous devices including	typewriters...tty(IV) interface
	typo(I) find possible typos
typo(I) find possible	typos
	umount(II) dismount file system
	umount(VIII) dismount file system
	uniq(I) report repeated lines in a file
mktemp(III) make a	unique named temporary file
load(VIII) load	/unix
rm(I) remove	(unlink) files
	unlink(II) remove directory entry
sync(II)	update super-block
sync(VIII)	update the super block

update(VIII) periodically	update the super block
	update(VIII) periodically update the super block
du(I) summarize disk	usage
mkpt(VIII) make prototype file for	use by mkfs
setuid(II) set process	user ID
getuid(II) get	user identifications
utmp(V)	user information
wtmp(V)	user login history
mail(I) send mail to designated	users
su(VIII) become privileged	user
wall(I) write to all	users
write(I) write to another	user
lseek(III) seek	using a long offset
	utmp(V) user information
abs, fabs(III) absolute	value
call, lcall,	vcall(II) create and execute a new process
fs(V) format of file system	volume
wait(II)	wait for process to terminate
	wait(I) await completion of process
	wait(II) wait for process to terminate
	wall(I) write to all users
	wc(I) word count
crash(VIII)	what to do when the system crashes
	what(I) identify SCCS files
who(I)	who is on the system
	who(I) who is on the system
telinit(VIII) superuser communication	with init
wc(I)	word count
hyphen(VI) find hyphenated	words
pwd(I)	working directory name
chdir, cd(I) change	working directory
chdir(II) change	working directory
putchar, flush(III)	write character
mesg(III)	write message on typewriter
write(II)	write on a file
wall(I)	write to all users
write(I)	write to another user
	write(I) write to another user
	write(II) write on a file
open(II) open for reading or	writing
	wtmp(V) user login history
	wump(VI) the game of hunt-the-wumpus
	yacc(I) yet another compiler-compiler
yacc(I)	yet another compiler-compiler

NAME

adb − debugger

SYNOPSIS

adb [−w] [objfil [corfil]]

DESCRIPTION

adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of UNIX programs.

objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of *adb* cannot be used although the file can still be examined. The default for *objfil* is **a.out.** *corfil* is assumed to be a core image file produced after executing *objfil;* the default for *corfil* is **core.**

Requests to *adb* are read from the standard input and responses are to the standard output. If the **−w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb. adb* ignores QUIT; INTERRUPT causes return to the next *adb* command.

In general requests to *adb* are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present then *dot* is set to *address.* Initially *dot* is set to 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *address* and *count* are expressions.

The interpretation of an address depends on the context it is used in. If a sub process is being debugged then addresses are interpreted in the usual way in the address space of the sub process. For further details of address mapping see **ADDRESSES.**

EXPRESSIONS

.	The value of *dot.*
+	The value of *dot* incremented by the current increment.
^	The value of *dot* decremented by the current increment.
"	The last *address* typed.
integer	An octal number if *integer* begins with a 0; a hexadecimal number if preceded by #; otherwise a decimal number.
integer. fraction	A 32 bit floating point number.
'cccc'	The ASCII value of up to 4 characters. \ may be used to escape a .
< name	The value of *name*, which is either a variable name or a register name. *adb* maintains a number of variables (q.v.) that are referred to by the letters **a** to **z** or the digits 0 to 9. If *name* is a register name then the value of the register is obtained from the system header in *corfil.* The register names are **r0 ... r5 sp pc ps**.
symbol	A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. \ may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil.* An initial _ or ~ will be prepended to *symbol* if needed.

routine . *name*	The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*.
(*exp*)	The value of *exp*.

Monadic operators

* *exp*	The contents of the location addressed by *exp* in *corfil*.
@ *exp*	The contents of the location addressed by *exp* in *objfil*.
− *exp*	Integer negation.
~ *exp*	Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1 + *e2*	Integer addition.
e1 − *e2*	Integer subtraction.
e1 * *e2*	Integer multiplication.
e1 % *e2*	Integer division.
e1 & *e2*	Bitwise conjunction.
e1 | *e2*	Bitwise disjunction.
e1 # *e2*	*e1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see **ADDRESSES** for further details.)

? *f*	Locations starting at *address* in *objfil* are printed according to the format *f*. *dot* is incremented by the sum of the increments for each format letter (q.v.).
/ *f*	Locations starting at *address* in *corfil* are printed according to the format *f* and *dot* is incremented as for '?'.
= *f*	The value of *address* itself is printed in the styles indicated by the format *f*. (For **i** format '?' is printed for the parts of the instruction that reference subsequent words.)

Formats

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format *dot* is incremented by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows.

o	2	Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long signed octal.
d	2	Print in decimal.
D	4	Print long decimal.

x	2	Print 2 bytes in hexadecimal.
X	4	Print 4 bytes in hexadecimal.
u	2	Print as an unsigned decimal number.
U	4	Print long unsigned decimal.
f	4	Print the 32 bit value as a floating point number.
F	8	Print double floating point.
b	1	Print the addressed byte in octal.
c	1	Print the addressed character.
C	1	Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s	*n*	Print the addressed characters until a zero character is reached.
S	*n*	Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.
Y	4	Print 4 bytes in date format (see *time (II)*).
i	*n*	Print as PDP11 instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a	0	Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below. / local or global data symbol ? local or global text symbol = local or global absolute symbol
p	2	Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.
t	0	When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8 space tab stop.
r	0	Print a space.
n	0	Print a newline.
"..."	0	Print the enclosed string.
^		*dot* is decremented by the current increment. Nothing is printed.
+		*dot* is incremented by 1. Nothing is printed.
−		*dot* is decremented by 1. Nothing is printed.

[?/] **l** *value mask*

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If **L** is used then the match is for 4 bytes at a time instead of 2. If no match is found then *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted then −1 is used.

[?/] w *value ...*
value is written into the addressed location. If **W** is used then 4 bytes are written, otherwise 2 bytes are written. Odd addresses are not allowed when writing to the sub process address space.

[?/] m *b1 e1 f1* **[?/]**
New values for (*b1*, *e1*, *f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If the list is terminated by '?' or '/' then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to *objfil*.)

> *name* *dot* is assigned to the variable or register named.

! A shell is called to read the rest of the line following '!'.

$ *modifier*

< *f* Read commands from the file *f* and return.

> *f* Send output to the file *f* which is created if it does not exist.

r Print the general registers and the instruction addressed by **pc**. *dot* is set to **pc**.

f Print the floating registers in single or double length. If the floating point status of **ps** is set to double (0200 bit) then double length is used anyway.

b Print all breakpoints and their associated counts and commands.

a ALGOL 68 stack backtrace. If *address* is given then it is taken to be the address of the current frame (instead of **r4**). If *count* is given then only the first *count* frames are printed.

c C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of **r5**). If **C** is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.

e The names and values of external variables are printed.

w Set the page width for output to *address* (default 80).

s Set the limit for symbol matches to *address* (default 255).

o All integers input are regarded as octal.

d Reset integer input as described in **EXPRESSIONS.**

q Exit from *adb.*

v Print all non zero variables in octal.

m The values used for mapping addresses into file addresses are printed.

: *modifier*

b *c* Set breakpoint at *address.* The breakpoint is executed *count*−1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.

d Delete breakpoint at *address.*

r Run *objfil* as a sub process. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the sub process may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

c *s* The sub process is continued with signal *s.* If *address* is given then the sub process is continued at this address. If no signal is specified then the signal that caused the sub process to stop is sent. Breakpoint skipping is the same as for **r**.

s *s* As for **c** except that the sub process is single stepped *count* times. If there is no current sub process then *objfil* is run as a sub process as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the sub process.

k The current sub process, if any, is terminated.

VARIABLES

adb provides a number of variables. Named variables are set initially by *adb* but are not used subsequently. Numbered variables are reserved for communication as follows.

0	The last value printed.
1	The last offset part of an instruction source.
2	The previous value of variable 1.

On entry the following are set from the system header in the *corfil.* If *corfil* does not appear to be a **core** file then these values are set from *objfil.*

b	The base address of the data segment.
d	The data segment size.
e	The entry point.
m	The 'magic' number (0405, 0407, 0410 or 0411).
s	The stack segment size.
t	The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples $(b1, e1, f1)$ and $(b2, e2, f2)$ and the *file address* corresponding to a written *address* is calculated as follows.

$$b1 \leqslant address < e1 \;=>\; file\ address = address + f1 - b1, \text{ otherwise,}$$

$$b2 \leqslant address < e2 \;=>\; file\ address = address + f2 - b2,$$

otherwise, the requested *address* is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a **?** or **/** is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, $b1$ is set to 0, $e1$ is set to the maximum file size and $f1$ is set to 0; in this way the whole file can be examined with no address translation.

So that *adb* may be used on large files all appropriate values are kept as signed 32 bit integers.

EXIT STATUS

If the last command was successful then zero; otherwise non zero.

BUGS

a) A breakpoint set at the entry point is not effective on initial entry to the program.

b) When single stepping, system calls do not count as an executed instruction.

c) Death of a child signals are seen by *adb* only when being caught by the process being run under *adb*.

FILES

/dev/mem
/dev/swap

SEE ALSO

ptrace (II)
a.out (V)
core (V)

NAME

admin – administer SCCS files

SYNOPSIS

admin [**−d**desc] [**−p**person] [**−t**type] [**−n**] [**−i**name [**−r**rel]] [**−c**ceil] [**−f**floor] [**−l** lock−rel]... [**−u**unlock−rel] ... [**−a**add−login] ... [**−e**erase−login] ... [**−y**[def−rel]] [**−z**] name ...

DESCRIPTION

Admin is used to create new SCCS files and change parameters of of existing ones. Arguments to *admin,* which may appear in any order, consist of keyletter arguments, which begin with "−", and named files. If a named file doesn't exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory was specified as a named file, except that non-SCCS files, as determined by their magic numbers (see *sccsfile* (V)), and unreadable files are silently ignored. If a name of "−" is given, the standard input is read; and each line of the standard input is taken to be the name of an SCCS file to be processed. Again non-SCCS files, and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument other than **i** and **r** apply independently to each named file.

−d	The description, up to 99 characters.
−p	The person, up to 13 characters.
−t	The type, up to 9 characters.
−n	This argument indicates that new files are to be created. This argument must be specified when creating new SCCS files. The **i** argument implies an **n** argument.
−i	The name of a file from which the text of an initial delta is taken. If this argument is supplied, but the file name is omitted, then the text is obtained by reading the standard input until an end−of−file is encountered. If this argument is omitted, and the *admin* command creates one or more SCCS files, then their initial deltas must be inserted in the normal manner, using *get* and *delta* (I). Only one SCCS file may be created by an *admin* command on which the **i** argument is supplied.
−r	The release into which the initial delta will be inserted. This argument may only be supplied if the **i** argument is also supplied. If this argument is omitted, the initial delta will be inserted into release 1. The level of the initial delta will always be 1.
−c	The "ceiling:" the highest release (less than or equal to 9999) which may be specified by the **r** argument on a *get* with an **e** argument. If omitted, the ceiling is 9999.
−f	The "floor:" the lowest release (greater than 0) which may be specified by the **r** argument on a *get* with an **e** argument. If omitted, the floor is 1.
−l	A release (between 1 and 9999, inclusive) to be locked. Several **l** arguments may be supplied on a single *admin* command, but at most 5 releases may be locked simultaneously. No one can add a delta to a locked release. Note that releases lower than the floor or higher than the ceiling are also effectively locked.

−u	A release (previously locked) to be unlocked. Several **u** arguments may be supplied on a single *admin* command.
−a	A login name to be added to the list of logins who may add deltas. Several **a** arguments may be supplied on a single *admin* command. As many logins as desired may be on the list simultaneously. If the list of logins is empty, then anyone may add deltas.
−e	A login name (or numerical user **ID**) to be erased from the list of logins. Several **e** arguments may be supplied on a single *admin* command.
−y	Default release if the **r** argument is omitted on a *get*. If no default release is specifically set with the **y** argument, or a null or zero value is set with the **y** argument, then the default release is the highest one which actually has a delta. If no releases have deltas, the default release is 1.
−z	The **z** keyletter is reserved for SCCS support personnel (those who instal and/or maintain SCCS) use only. It should not be used by anyone else.

FILES

Created files have mode 444. Write permission (either explicitly or implicitly) in the pertinent directory is, of course, required to create a file. If an existing file is not writable, an attempt will be made to change its mode to 644 during execution of *admin,* after which its mode will be restored to what it was before execution. Note that the mode can be changed only if th effective user owns the file.

It is recommended that directories containing SCCS files be mode 555 (commands which need to create or remove files will attempt to change the mode of the directory to be writable and searchable, and will afterwards restore the mode to what it was; note that the mode can be changed only if the effective user owns the directory), and that SCCS files themselves be mode 444 (see *chmod* (I)).

SEE ALSO

chmod(I), get(I), delta(I), prt(I),what(I), help(I), sccsfile(V)
SCCS/PWB User's Manual
The Source Code Control System

DIAGNOSTICS

Use *help* (I) for explanations.

NAME

ar — archive and library maintainer

SYNOPSIS

ar key [posname] afile [name ...]

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose.

Key is one character from the command set **drtlpmx** optionally concatenated with characters from the option set **vunab.** *Afile* is the archive file. The *names* are constituent files in the archive file. The *posname* is the name of a file in the archive; it is required if and only if one of the options **a** or **b** is used in the *key*. The meanings of the *key* characters are:

d Delete the named files from the archive file.

r Replace or add the named files. New files are placed at the end of the archive unless the **a** or **b** option is used. Existing files are replaced in their current position and **a** or **b** have no effect.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled.

l List the contents of the archive in long mode, giving mode, owner, size in bytes and time of last modification for each file. If no names are given, all files are listed.

p Print the named files in the archive.

m Move the named files to the end of the archive or as specified by the **a** or **b** option.

x Extract the named files. If no names are given, all files in the archive are extracted. If the optional character **u** is used with **x,** only those files with a modified date earlier than the dates of the respective files within the archive will be replaced by extracted files.

v Verbose; under the verbose option, *ar* gives additional information while working:

d,r,m	gives log of changes to archive.
x	logs files extracted.
t,l	changes **t** command into **l** command.
p	prints name of file before printing file.

u Update; under the update option *ar,* in replacing or extracting, will only over-write an older version of a file, if versions exist both within and without the archive (see **r** and **x** command descriptions).
If **u** is part of a *key* with no command character, **r** is included as the command.

n Interactive; under the interactive option, *ar* requests information from the standard input before modifying the archive.

- **r** before creating the archive, *ar* prints:
  ? - afile?
  Reply 'y' to create the archive; any other response and *ar* will abort.

- **m** before modifying the archive, *ar* prints:
  n - filename?
  Reply with a new name to rename the file; an illegal filename or just a return will retain the old name.

- **d** before deleting any file, *ar* prints:
  d - filename?
  Reply 'y' to delete the file; any other reply and the file will be kept.

a,b rearranges files in the archive file, placing them after or before the file designated as *posname*. These options require that *posname* follow the key.

FILES

/tmp/vtm? temporary

SEE ALSO

ld (I), archive (V)

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice. The options **a** and **b** should work with **r** when the file already exists in the archive.

AS(I) AS(I)

NAME

as − assembler

SYNOPSIS

as [−] [**−o ofile**] name ...

DESCRIPTION

As assembles the concatenation of the named files. If the optional first argument − is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file **a.out.** If the optional argument **−o** followed by a file name is given, the output will be left on that file without affecting a previously existing **a.out** file. It is executable if no errors occurred during the assembly, and if there were no unresolved external references.

FILES

/lib/as2	pass 2 of the assembler
/tmp/atm[1-3]?	temporary
a.out	object

SEE ALSO

ld (I), nm (I), adb (I), strip (I), a.out (V), 'UNIX Assembler Manual'.

DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
]	Parentheses error
<	String not terminated properly
*	Indirection used illegally
.	Illegal assignment to '.'
A	Error in address
B	Branch instruction is odd or too remote
E	Error in expression
F	Error in local ('f' or 'b') type symbol
G	Garbage (unknown) character
I	End of file inside an if
M	Multiply defined symbol as label
O	Word quantity assembled at odd address
P	'.' different in pass 1 and 2
R	Relocation error
U	Undefined symbol
X	Syntax error

BUGS

Symbol table overflow is not checked. **X** errors can cause incorrect line numbers in following diagnostics.

NAME

bas − basic

SYNOPSIS

bas [file]

DESCRIPTION

Bas is a dialect of Basic. If a file argument is provided, the file is used for input before the console is read. *Bas* accepts lines of the form:

statement
integer statement

Integer numbered statements (known as internal statements) are stored for later execution. They are stored in sorted ascending order. Non-numbered statements are immediately executed. The result of an immediate expression statement (that does not have '=' as its highest operator) is printed.

Statements have the following syntax:

expression

The expression is executed for its side effects (assignment or function call) or for printing as described above.

done

Return to system level.

draw expression expression expression

A line is drawn on the Tektronix 611 display '/dev/vt0' from the current display position to the XY co-ordinates specified by the first two expressions. The scale is zero to one in both X and Y directions. If the third expression is zero, the line is invisible. The current display position is set to the end point.

display list

The list of expressions and strings is concatenated and displayed (i.e. printed) on the 611 starting at the current display position. The current display position is not changed.

erase

The 611 screen is erased.

for name = expression expression statement
for name = expression expression
...
next

The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression.

goto expression

The expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statment. If executed from immediate mode, the internal statements are compiled first.

if expression statement

The statement is executed if the expression evaluates to non-zero.

list [expression [expression]]

is used to print out the stored internal statements. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed.

print list
The list of expressions and strings are concatenated and printed. (A string is delimited by double quotes.)

return [expression]
The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run
The internal statements are compiled. The symbol table is re-initialized. The random number generator is reset. Control is passed to the lowest numbered internal statement.

Expressions have the following syntax:

name
A name is used to specify a variable. Names are composed of a letter followed by letters and digits. The first four characters of a name are significant.

number
A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an e followed by an optionally signed exponent.

(expression)
Parentheses are used to alter normal order of evaluation.

expression operator expression
Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. A complete list of operators is given below.

expression ([expression [, expression] ...])
Functions of an arbitrary number of arguments can be called by an expression followed by the arguments in parentheses separated by commas. The expression evaluates to the line number of the entry of the function in the internally stored statements. This causes the internal statements to be compiled. If the expression evaluates negative, a built-in function is called. The list of built-in functions appears below.

name [expression [, expression] ...]
Each expression is truncated to an integer and used as a specifier for the name. The result is syntactically identical to a name. **a[1,2]** is the same as **a[1][2].** The truncated expressions are restricted to values between 0 and 32767.

The following is the list of operators:

=

= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

& |

& (logical and) has result zero if either of its arguments are zero. It has result one if both its arguments are non-zero. | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments are non-zero.

< <= > >= == <>

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, <> not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: a>b>c is the same as a>b&b>c.

+ −

Add and subtract.

* /

Multiply and divide.

Exponentiation.

The following is a list of built-in functions:

arg(i)
is the value of the *i*-th actual parameter on the current level of function call.

exp(x)
is the exponential function of *x*.

log(x)
is the natural logarithm of *x*.

sin(x)
is the sine of *x* (radians).

cos(x)
is the cosine of *x* (radians).

atn(x)
is the arctangent of *x*. Its value is between $-\pi/2$ and $\pi/2$.

rnd( )
is a uniformly distributed random number between zero and one.

expr( )
is the only form of program input. A line is read from the input and evaluated as an expression. The resultant value is returned.

int(x)
returns *x* truncated to an integer.

FILES

/tmp/btm? temporary

DIAGNOSTICS

Syntax errors cause the incorrect line to be typed with an underscore where the parse failed. All other diagnostics are self explanatory.

BUGS

Has been known to give core images. Needs a way to *list* a program onto a file.

NAME

bc − arbitrary precision interactive language

SYNOPSIS

bc [**−l**] [file ...]

DESCRIPTION

Bc is an interactive processor for a language which resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The '−l' argument stands for the name of a library of mathematical subroutines which contains sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), and exponential ('e'). The syntax for *bc* programs is as follows; E means expression, S means statement.

Comments
- are enclosed in /* and */.

Names
- letters a−z
- array elements: letter[E]
- the words 'ibase', 'obase', and 'scale'

Other operands
- arbitrarily long numbers with optional sign and decimal point
- (E)
- sqrt (E)
- <letter> (E , ... , E)

Operators
```
+  −  *  /  %  ^
++  −−        (prefix and postfix; apply to names)
==  <=  >=  !=  <  >
=  =+  =−  =*  =/  =%  =^
```

Statements
```
E
{ S ; ... ; S }
if ( E ) S
while ( E ) S
for ( E ; E ; E ) S
null statement
break
quit
```

Function definitions are exemplified by
```
define <letter> ( <letter> ,..., <letter> ) {
        auto <letter>, ... , <letter>
        S; ... S
        return ( E )
}
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations. Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array name, a function name, and a simple variable simultaneously. 'Auto' variables are saved and restored during function calls. All other variables are global to the program. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

For example

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1; i++){
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
        for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

/usr/lib/lib.b   mathematical library

SEE ALSO

dc (I), C Reference Manual, ''BC – An Arbitrary Precision Desk-Calculator Language.''

BUGS

No &&, || yet.
for statement must have all three E's
quit is interpreted when read, not when executed.

CAT(I) CAT(I)

NAME

cat – concatenate and print

SYNOPSIS

cat [**−u**] file ...

DESCRIPTION

Cat reads each file in sequence and writes it on the standard output. Thus

cat file

prints the file, and

cat file1 file2 >file3

concatenates the first two files and places the result on the third.

If the optional **−u** argument is present, the output is unbuffered. That is, a write occurs every wakeup as opposed to every 512 bytes.

If no input file is given, or if the argument '−' is encountered, *cat* reads from the standard input file.

SEE ALSO

pr (I), cp (I)

DIAGNOSTICS

"cat: can't open *file"* if *file* cannot be found. The file is ignored and **cat** continues.

BUGS

cat x y >x and **cat x y >y** should be avoided.

NAME

cc − C compiler

SYNOPSIS

cc [−c] [−p] [−f] [−O] [−S] [−P] [−E] [−Dsymbol ...] [−Iprefix] [−Cprefix] file ...

DESCRIPTION

Cc is the UNIX C compiler. It accepts three types of arguments:

Arguments whose names end with '.c' are taken to be C source programs and those with '.s' are taken as assembly programs; they are compiled/assembled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c' or '.s'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

The following flags are interpreted by *cc.* See *ld* (I) for load-time flags.

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls the *monitor* (III) subroutine at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof* (I).

−f In systems without hardware floating-point, use a version of the C compiler which handles floating-point constants and loads the object program with the floating-point interpreter. Do not use if the hardware is present. This does not permit cross machine compilations from floating to non-floating point.

−O Invoke an object-code optimizer.

−S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

−P Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed '.i'.

−E Same as the **−P** option except the output is directed to the standard output. This allows the preprocessor to be used as a filter for any other compiler.

-D Define the symbol **symbol** to the preprocessor. This mechanism is useful with the conditional statements in the preprocessor by allowing symbols to be defined external to the source file.

−I Use the path name **prefix** rather than the default **/usr/include** as the prefix directory to angle bracketed include files. Include statements with angle brackets (#include <file>) as opposed to quotes will have the prefix directory added to the name to form the referenced file name.

−C Use the string **prefix** rather than the default string **/compool/** for include statements. If the file name in a quoted include statement does not exist, the prefix is automatically combined with the include file name and this resulting file is included if it exists.

−h Changes the home directory for include statements to the current directory. Quoted **include** statements are normally relative to the location of the file its contained in as opposed to the current directory.

Other arguments are taken to be either loader flag arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out** unless the **-o** option of the loader is used.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/c[01]	compiler
/lib/fc[01]	floating-point compiler
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff of profiling
/lib/fcrt0.o	runtime startoff for floating-point interpretation
/lib/libc.a	C library; see section III.
/lib/liba.a	Assembler library used by some routines in libc.a

SEE ALSO

''Programming in C— a tutorial,'' C Reference Manual, monitor (III), prof (I), adb (I), ld (I).

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular ''m,'' which means a multiply-defined external symbol (function or data).

BUGS

CHDIR(I) CHDIR(I)

NAME

chdir, cd − change working directory

SYNOPSIS

chdir directory
cd directory

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *chdir* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh (I), pwd (I)

BUGS

CHMOD(I) CHMOD(I)

NAME

chmod − change mode

SYNOPSIS

chmod octal file ...

DESCRIPTION

The octal mode replaces the mode of each of the files. The mode is constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit for shared, pure-procedure programs (see below)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

Only the owner of a file (or the super-user) may change its mode.

If an executable file is set up for sharing (''−n'' option of *ld (I)*), then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

SEE ALSO

ls (I), chmod (II)

BUGS

CHROOT(I) CHROOT(I)

NAME

chroot − change root directory for a command

SYNOPSIS

chroot newroot command

DESCRIPTION

The given command is executed *relative to the new root.* The meaning of any initial slashes ("/") in path names is changed for a command and any of its children to **newroot.** Furthermore, the initial working directory is **newroot.**

Notice that **chroot newroot command > x** will create the file x relative to the original root, not the new one.

This command is restricted to the super user.

The new root pathname is always absolute: even if a *chroot* is currently in effect, the **newroot** argument is relative to the real root of the running system.

SEE ALSO

chroot(II)

BUGS

It is unwise and generally disastrous to reference special files in the new root file system.

CMP (I) CMP (I)

NAME

cmp − compare two files

SYNOPSIS

cmp [**−s**] [**−l**] file1 file2

DESCRIPTION

The two files are compared. Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted. Moreover, return code 0 is yielded for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

The optional argument **−s** is used to make a silent compare and only set the return code. The optional argument **−l** prints all differences in octal and can be used to compare binary files.

SEE ALSO

diff (I), comm (I)

BUGS

NAME

comm − print lines common to two files

SYNOPSIS

comm [**−** [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2,* which should be sorted, and produces a three column output: lines only in *file1;* lines only in *file2;* and lines in both files. The filename '−' means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm −12** prints only the lines common to the two files; **comm −23** prints only lines in the first file but not in the second; **comm −123** is a no-op.

SEE ALSO

cmp (I), diff (I)

BUGS

CP(I) CP(I)

NAME

cp – copy

SYNOPSIS

cp [−t] file1 file2

DESCRIPTION

The first file is copied onto the second. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If *file2* is a directory, then the target file is a file in that directory with the file-name of *file1*.

It is forbidden to copy a file onto itself.

The optional *−t* argument will report the number of 512 byte blocks that have been copied.

SEE ALSO

cat (I), pr (I), mv (I)

BUGS

CPALL(I) CPALL(I)

NAME

cpall – copy all files to a directory

SYNOPSIS

cpall name [...] dir

DESCRIPTION

The named files are copied into directory *dir* by means of successive calls to the command *cp(I)*. These calls are of the form

```
cp name dir/.
```

SEE ALSO

cp(I)

BUGS

Directories and special files are copied just as if they were ordinary files.

NAME

cref — make cross reference listing

SYNOPSIS

cref [−acilostux123] name ...

DESCRIPTION

Cref makes a cross reference listing of program files in assembler or C format. The files named as arguments in the command line are searched for symbols in the appropriate syntax.

The output report is in four columns:

(1)	(2)	(3)	(4)
symbol	file	see below	text as it appears in file

Cref may use either an *ignore* file or an *only* file. If the **−i** option is given, the next argument is taken to be an *ignore* file; if the **−o** option is given, the next argument is taken to be an *only* file. *Ignore* and *only* files are lists of symbols separated by new lines. All symbols in an *ignore* file are ignored in columns (1) and (3) of the output. If an *only* file is given, only symbols in that file appear in column (1). At most one of **−i** and **−o** may be used. The default setting is **−i.** Assembler predefined symbols or C keywords are ignored.

The **−s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, it is the current function name. The **−l** option causes the line number within the file to be put in column 3.

The **−t** option causes the next available argument to be used as the name of the intermediate temporary file (instead of /tmp/crt??). The file is created and is not removed at the end of the process. The format of the output under this option is not as above.

Options:

a	assembler format (default)
c	C format input
i	use *ignore* file (see above)
l	put line number in col. 3 (instead of current symbol)
o	use *only* file (see above)
s	current symbol in col. 3 (default)
t	user supplied temporary file
u	print only symbols that occur exactly once
x	print only C external symbols
1	sort output on column 1 (default)
2	sort output on column 2
3	sort output on column 3

FILES

/tmp/crt??	temporaries
/usr/lib/aign	default assembler *ignore* file
/usr/lib/cign	default C *ignore* file
/usr/bin/crpost	post processor
/usr/bin/upost	post processor for **−u** option
/bin/sort	used to sort temporaries

SEE ALSO

as (I), cc (I)

BUGS

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]]

DESCRIPTION

If no argument is given, the current date is printed to the second. If an argument is given(as super user), the current date is set and the old time and new time are stamped into the */etc/wtmp* file (if it exists). The first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

FILES

/etc/wtmp (if it exists)

BUGS

It is difficult to remain current with the rules for DST.

NAME

dc − desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number	The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.
+ − * % ^	The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.
s*x*	The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed on it.
l*x*	The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.
d	The top value on the stack is duplicated.
p	The top value on the stack is printed. The top value remains unchanged.
f	All values on the stack and in registers are printed.
q	exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.
x	treats the top element of the stack as a character string and executes it as a string of dc commands.
[...]	puts the bracketed ascii string onto the top of the stack.
<*x* >*x* =*x*	The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation.
v	replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.
!	interprets the rest of the line as a UNIX command.
c	All values on the stack are popped.
i	The top value on the stack is popped and used as the number radix for further input.
o	The top value on the stack is popped and used as the number radix for further output.
k	the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z	The stack level is pushed onto the stack.
?	A line of input is taken from the input source (usually the console) and executed.

An example which prints the first ten values of n! is

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc (I), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

(x) ? for unrecognized character x.
(x) ? for not enough elements on the stack to do what was asked by command x.
'Out of space' when the free list is exhausted (too many digits).
'Out of headers' for too many numbers being kept around.
'Out of pushdown' for too many items on the stack.
'Nesting Depth' for too many levels of nested execution.

BUGS

NAME

dd − convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	*values*
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=	input block size (default 512)
obs=	output block size (default 512)
bs=	set both input and output block size, superseding *ibs* and *obs;* also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
... , ...	several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **k, b** or **w** to specify multiplication by 1024, 512, or 2 respectively. Also a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs.*

After completion, *dd* reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x:*

```
dd  if=/dev/rmt0  of=x  ibs=800  cbs=80  conv=ascii,lcase
```

Note the use of raw magtape. *Dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

SEE ALSO

cp (I)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. It is not clear how this relates to real life.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. There should be separate options.

DELTA(I) DELTA(I)

NAME

delta – make an SCCS delta

SYNOPSIS

delta [–s] [–n] [–alist] [–yhistory] [–l] [–h]

DESCRIPTION

Delta adds a *delta* to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files, as determined by their magic numbers (see *sccsfile* (V)), and unreadable files are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored.

A *get* of many SCCS files, followed by a *delta* of those files should be avoided when the *get* generates a large amount of data. Instead, multiple *get* – *delta* sequences should be used.

The **a** argument specifies a list (see *get* for the definition of <list>) of heretofore unacknowledged deltas which are to be marked acknowledged. An unacknowledged delta should only be acknowledged when the problem that caused the creation of the unacknowledged (non-propagating) delta is fixed by another delta.

A comment about the purpose of the delta(s) is supplied (once, and only once) either from the standard input, or by using the **y** argument. If one supplies the comment through the standard input, and the standard input is a terminal (as determined by a successful *gtty* (II) call), the program will prompt (on the standard output) with "history?". Otherwise, no prompt is printed. A newline preceded by a "\" is read as a blank, and may be used to make the entering of the history more convenient. The first newline not preceded by a "\" terminates the history response. The **y** argument is used to supply history on the command line; if it is given the "history?" question is not printed, and the standard input is not read.

The following description is written as though only one SCCS file were named; the process of making a delta is equivalent for each file. (Note that the effects of any keyletter arguments apply independently to each SCCS file, and that the same history comment is used for all files.)

Delta makes a delta by "getting" the named file (see *get*(I)) at the same release that was used when the *get* command was last executed with the **e** argument. The "gotten" file is then compared with the *g-file*; the differences between the two files constitute the delta. The new delta has a level number one greater than the original generation.

If the delta is made in a release which is lower than the highest numbered release that has deltas, then the delta is marked non-propagating (and unacknowledged). When the named file is "gotten" at any release higher than the one which contains the non-propagating delta, the non-propagating delta will *not* be applied. Also, until the delta is acknowledged (see below), *get*(I) will note it as being non-propagating.

When the comparison is finished, *delta* prints the release and level of the new delta, followed by the number of lines inserted, deleted, and unchanged. The **s** argument suppresses this printing. Normally, the *g-file* is removed after the delta is made. The **n** argument suppresses the removal.

Delta will ignore hangups if it is already ignoring interrupts.

The **l** and **h** arguments are reserved for SCCS support personnel (those who install and/or maintain SCCS) use only. They should not be used by anyone else.

FILES

g-file	see *get*(I) for explanation of *g-file*
p-file	Information from *get*
x-file	Replacement for the SCCS file. The naming convention is the same as that for the *p-file* (see *get*).

SEE ALSO

get(I), prt(I), help(I), sccsfile(V)
SCCS/PWB User's Manual, The Source Code Control System (which should not be taken to literally).

DIAGNOSTICS

Use *help*(I) for explanations.

BUGS

The acknowledged/unacknowledged non-propagating delta mechanism is a kludge. One has to make a delta in order to acknowledge some other delta. The acknowledge "bit" is part of the delta, and is *not* a function of the release being accessed.

DIFF(I) DIFF(I)

NAME

diff − differential file comparator

SYNOPSIS

diff [**−efb**] name1 name2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If **name1 (name2)** is '−', the standard input is used. If **name1 (name2)** is a directory, then a file in that directory whose file-name is the same as the file-name of **name2 (name1)** is used. The normal output contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble *ed* commands to convert file **name1** into file **name2.** The numbers after the letters pertain to file **name2.** In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert file **name2** into **name1.** As in *ed,* identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

The **-b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal. The **−e** option produces a script of *a, c* and *d* commands for the editor *ed,* which will recreate file **name2** from file **name1.** The **−f** option produces the same editing commands in the opposite order (not directly usable by *ed.*) In connection with **-e,** the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(cat $2 ... $9; echo 1,$p) | ed − $1
```

Except for occasional 'jackpots', *diff* finds a smallest sufficient set of file differences.

FILES

/tmp/d?????

SEE ALSO

cmp (I), comm (I), ed (I)

DIAGNOSTICS

'jackpot' − To speed things up, the program uses hashing. You have stumbled on a case where there is a chance that this has resulted in a difference being called where none actually existed. Sometimes reversing the order of files will make a jackpot go away.

BUGS

Editing scripts produced under the **−e** option are naive about creating lines consisting of a single '.'.

DIFF3(I) DIFF3(I)

NAME

diff3 − 3-way differential file comparison

SYNOPSIS

diff3 [**−ex3**] file1 file2 file3

DESCRIPTION

Diff3 Compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1*	Text is to be appended sfter line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1,n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range is given immediately after a **c** indication. To shorten the output when the original contents of two files are flagged as identical, the contents of the lower-numbered file is suppressed.

Under the **−e** option, *diff3* publishes an editing script that will incorporate into *file1* all changes between *file2* and *file3, i.e.* the changes that normally would be flagged ==== and ====3. Option **−x** (−3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to 'file1'.

```
(cat script; echo "1,$p") | ed - file1
```

FILES

d3*p*[12], where *p* is a process number: temporaries

SEE ALSO

diff(I)

BUGS

Under **−e**, text lines that consist of a single '.' will bomb.

NAME

dsw – delete interactively

SYNOPSIS

dsw [directory]

DESCRIPTION

For each file in the given directory ('.' if not specified) *dsw* types its name. If **y** is typed, the file is deleted; if **x**, *dsw* exits; if new-line, the file is not deleted; if anything else, *dsw* asks again.

SEE ALSO

rm (I)

BUGS

The name *dsw* is a carryover from the ancient past.

DU (I)

NAME

du – summarize disk usage

SYNOPSIS

du [**−s**] [**−a**] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file *name.* If *name* is missing, '**.**' is used.

The optional argument **−s** causes only the grand total to be given. The optional argument **−a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

A file which has two links to it is only counted once.

BUGS

Non-directories given as arguments (not under **−a** option) are not listed.

Removable file systems do not work correctly since i-numbers may be repeated while the corresponding files are distinct. *Du* should maintain an i-number list per root directory encountered.

NAME

echo − echo arguments

SYNOPSIS

echo [arg ...]

DESCRIPTION

Echo writes its arguments in order as a line on the standard output file. It is mainly useful for producing diagnostics in command files.

If the last character of a line to be echoed is followed by a space (which must be quoted), then the echoed line will not be followed by a newline.

BUGS

NAME

ed – text editor

SYNOPSIS

ed [–] [name **]**

DESCRIPTION

Ed is the standard text editor.

If a **name** argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed's* buffer so that it can be edited. The optional – suppresses the printing of character counts by *e, r,* and *w* commands.

Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer.* There is only one buffer.

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command,* possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Every command which requires addresses has default addresses, so that the addresses can often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode.* In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period '.' alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation. A regular expression specifies a set of strings of characters. A particular regular expression *matches* a string of characters when a line of text can be found that contains one of the desired character strings. The regular expressions allowed by *ed* are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.
2. A circumflex ' ' at the beginning of a regular expression matches the empty string at the beginning of a line.
3. A currency symbol '$' at the end of a regular expression matches the null character at the end of a line.
4. A period '.' matches any character except a new-line character.
5. A regular expression followed by an asterisk '*' matches any number of adjacent occurrences (including zero) of the regular expression it follows.
6. A string of characters enclosed in square brackets '[ ]' matches any character in the string but no others. An inclusive sequence of characters can be indicated with two characters separated with '-'. If, however, the first character of the string is a circumflex ' ', the regular expression matches any character except new-line and the characters in the string.
7. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.
8. A regular expression enclosed between the sequences '\(' and '\)'is identical to the unadorned expression; the construction has side effects discussed under the *s* command, and section 9. just following.

9. The expression ‘\n’ means the same string of characters matched by an expression enclosed in ‘\\(’ and ‘\\)’ earlier in the same whole expression. Here *n* is a digit; the subexpression specified is that beginning with the *n*-th occurrence of ‘\\(’ counting from the left. For example the expression ‘ \\(.*\\)\\1$’ matches a line consisting of two repeated appearances of the same string.

10. The null regular expression standing alone is equivalent to the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by ‘\\’. This also applies to the character bounding the regular expression (often ‘/’) and to ‘\\’ itself.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line*. Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character ‘.’ addresses the current line.
2. The character ‘$’ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. ‘′*x*’ addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A regular expression enclosed in slashes ‘/’ addresses the first line found by searching toward the end of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the beginning of the buffer.
6. A regular expression enclosed in queries ‘?’ addresses the first line found by searching toward the beginning of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the end of the buffer.
7. An address followed by a plus sign ‘+’ or a minus sign ‘−’ followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with ‘+’ or ‘−’ the addition or subtraction is taken with respect to the current line; e.g. ‘−5’ is understood to mean ‘.−5’.
9. If an address ends with ‘+’ or ‘−’, then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address ‘−’ refers to the line before the current line. Moreover, trailing ‘+’ and ‘−’ characters have cumulative effect, so ‘−−’ refers to the current line less 2.
10. To maintain compatibility with earlier versions of the editor, the character ‘ ’ in addresses is entirely equivalent to ‘−’.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma ','. They may also be separated by a semicolon ';'. In this case the current line '.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, any command may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below.

(.)a
<text>
.

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

(. , .)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. '.' is left at the last line input; if there were none, it is left at the first line not deleted.

(. , .) d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '.' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent *r* or *w* command. An error will occur if prior buffer modifications have not been written with the *w* command. A subsequent *e* command will ignore the modifications.

E filename

This command performs the same function as the *e* command except no error will result if buffer modifications have been made.

f filename

The filename command prints the currently remembered file name. If 'filename' is given, the currently remembered file name is changed to 'filename'.

(1,$) g/regular expression/command list

In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. The *a, i,* and *c* commands and associated input are permitted; the '.' terminating input mode may be omitted if it would be on the last line of the command list. The (global) commands, *g* and *v*, are not permitted in the command list.

(.) i
<text>

This command inserts the given text before the addressed line. '.' is left at the last line input; if there were none, at the addressed line. This command differs from the *a* command only in the placement of the text.

(.) k*x*

The mark command marks the addressed line with name *x*, which must be a lower-case letter. The address form ''*x*' then addresses this line.

(. , .) l

The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in octal, and long lines are folded. An *l* command may follow any other on the same line.

(. , .) m*a*

The move command repositions the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

(. , .) p

The print command prints the addressed lines. '.' is left at the last line printed. The *p* command may be placed on the same line after any command.

q

The quit command causes *ed* to exit. No automatic write of a file is done. An error will occur if prior buffer modifications have not been written with the *w* command. A subsequent *q* command will ignore the modifications and exit.

Q

This command performs the same function as the *q* command except no error will result if buffer modifications have been made.

($) r filename

The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The remembered file name is not changed unless 'filename' is the very first file name mentioned. Address '0' is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '.' is left at the last line read in from the file.

(. , .) s/regular expression/replacement/ or,
(. , .) s/regular expression/replacement/g

The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of '/' to delimit the regular expression and the replacement. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context may be suppressed by preceding it by '\'. As a more general feature, the characters '*n*', where *n* is a digit, are replaced by the text matched by the *n*-th regular subexpression enclosed between '\(' and '\)'. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by '\'.

(. , .) t *a*

This command acts just like the *m* command, except that a copy of the addressed lines is placed after address *a* (which may be 0). '.' is left on the last line of the copy.

(.) u

This command is used to undo the effect of the last substitute *s* command. The line of text will be restored to its state prior to the last substitute command. The addressed line must be identical to the line the substitute was performed on.

(1,$) v/regular expression/command list

This command is the same as the global command except that the command list is executed with '.' initially set to every line *except* those matching the regular expression.

(1,$) w filename

The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writeable by everyone). The remembered file name is *not* changed unless 'filename' is the very first file name mentioned. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). '.' is unchanged. If the command is successful, the number of characters written is typed.

($) =

The line number of the addressed line is typed. '.' is unchanged by this command.

!UNIX command

The remainder of the line after the '!' is sent to UNIX to be interpreted as a command. '.' is unchanged.

(.+1) <newline>

An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, *ed* prints a '?' and returns to its command level.

The following size limitations apply: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 1 word.

FILES

/tmp/e#, temporary; '#' is the process number (in octal).

DIAGNOSTICS

'?' for errors in commands; 'TMP' for temporary file overflow; 'MEM' for memory overflow (too many lines); 'WRITE ERROR' if a physical I/O error occurs while attempting to write a file.

SEE ALSO

A Tutorial Introduction to the ED Text Editor, B. W. Kernighan.
Advanced Editing on UNIX, B. W. Kernighan.

BUGS

The *s* command causes all marks to be lost on lines changed.

NAME

eqn — typeset mathematics

SYNOPSIS

eqn [file] ...

DESCRIPTION

Eqn is a troff (I) preprocessor for typesetting mathematics on the Graphics Systems phototypesetter. Usage is almost always

eqn file ... | troff

This manual page was piped through *eqn* in the way shown above.

If no files are specified, *eqn* reads from the standard input. A line beginning with ‘‘.EQ’’ marks the start of an equation; the end of an equation is marked by a line beginning with ‘‘.EN’’. Neither of these lines is altered or defined by *eqn,* so you can define them yourself to get centering, numbering, etc. All other lines are treated as comments, and passed through untouched.

Spaces, tabs, newlines, braces, double quotes, tilde and circumflex are the only delimiters. Braces ‘‘{}’’ are used for grouping. Use tildes ‘‘~’’ to get extra spaces in an equation.

Subscripts and superscripts are produced with the keywords **sub** and **sup.** Thus *x sub i* makes x_i, *a sub i sup 2* produces a_i^2, and *e sup {x sup 2 + y sup 2}* gives $e^{x^2+y^2}$. Fractions are made with **over.** *a over b* is $\frac{a}{b}$ and *1 over sqrt {ax sup 2 +bx+c}* is $\frac{1}{\sqrt{ax^2+bx+c}}$. **sqrt** makes square roots.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $\lim_{n \to \infty} \sum_{0}^{n} x_i$ is made with *lim from {n-> inf} sum from 0 to n x sub i.* Left and right brackets, braces, etc., of the right height are made with **left** and **right:** *left [x sup 2 + y sup 2 over alpha right] ~=~1* produces $\left[x^2 + \frac{y^2}{\alpha} \right] = 1$. The **right** clause is optional.

Vertical piles of things are made with **pile, lpile, cpile,** and **rpile:** *pile {a above b above c}* produces $\begin{matrix} a \\ b \\ c \end{matrix}$. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile** center, with different vertical spacing, and **rpile** right justifies.

Diacritical marks are made with **dot, dotdot, hat, bar:** *x dot = f(t) bar* is $\dot{x} = \overline{f(t)}$. Default sizes and fonts can be changed with **size n** and various of **roman, italic,** and **bold.**

Keywords like *sum* ($\sum$) *int* ($\int$) *inf* (∞) and shorthands like >= ($\geqslant$) -> ($\rightarrow$), != ($\neq$), are recognized. Spell out Greek letters in the desired case, as in *alpha, GAMMA.* Mathematical words like sin, cos, log are made Roman automatically. Troff (I) four-character escapes like \(bs (🔔) can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched.

SEE ALSO

A System for Typesetting Mathematics (Computer Science Technical Report #17, Bell Laboratories, 1974.)
TROFF Users' Manual (internal memorandum)
TROFF Made Trivial (internal memorandum)
troff (I), neqn (I)

BUGS

Undoubtedly. Watch out for small or large point sizes – it's tuned too well for size 10. Be cautious if inserting horizontal or vertical motions, and of backslashes in general.

NAME

exit − terminate command file

SYNOPSIS

exit [rtncode]

DESCRIPTION

Exit performs a **seek** to the end of its standard input file. Thus, if it is invoked inside a file of commands, upon return from *exit* the shell will discover an end-of-file and terminate. The optional numeric argument *rtncode,* if given, is used by *exit* for the return code; otherwise the return code is set to 0.

SEE ALSO

if (I), goto (I), sh (I)

BUGS

NAME

fc − Fortran compiler

SYNOPSIS

fc [**−c**] sfile1.f ... ofile1 ...

DESCRIPTION

Fc is the UNIX Fortran compiler. It accepts three types of arguments:

Arguments whose names end with '.f' are assumed to be Fortran source program units; they are compiled, and the object program is left on the file sfile1.o (i.e. the file whose name is that of the source with '.o' substituted for '.f').

Other arguments (except for **−c**) are assumed to be either loader flags, or object programs, typically produced by an earlier *fc* run, or perhaps libraries of Fortran-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

The **−c** argument suppresses the loading phase, as does any syntax error in any of the routines being compiled.

The following is a list of differences between *fc* and ANSI standard Fortran (also see the BUGS section):

1. Arbitrary combination of types is allowed in expressions. Not all combinations are expected to be supported at runtime. All of the normal conversions involving integer, real, double precision and complex are allowed.
2. Two forms of "implicit" statements are recognized: **implicit integer /i−n/** or **implicit integer (i−n).**
3. The types doublecomplex, logical*1, integer*1, integer*2, integer*4 (same as integer), real*4 (real), and real*8 (double precision) are supported.
4. **&** as the first character of a line signals a continuation card.
5. **c** as the first character of a line signals a comment.
6. All keywords are recognized in lower case.
7. The notion of 'column 7' is not implemented.
8. G-format input is free form− leading blanks are ignored, the first blank after the start of the number terminates the field.
9. A comma in any numeric or logical input field terminates the field.
10. There is no carriage control on output.
11. A sequence of *n* characters in double quotes '"' is equivalent to *n* **h** followed by those characters.
12. In **data** statements, a hollerith string may initialize an array or a sequence of array elements.
13. The number of storage units requested by a binary **read** must be identical to the number contained in the record being read.
14. If the first character in an input file is "#", a preprocessor identical to the C preprocessor is called, which implements "#define" and "#include" preprocessor statements. (See the C reference manual for details.) The preprocessor does not recognize Hollerith strings written with *n.*

In I/O statements, only unit numbers 0-19 are supported. Unit number *n* refers to file fort*nn;* (e.g. unit 9 is file 'fort09'). For input, the file must exist; for output, it will be created. Unit 5 is permanently associated with the standard input file; unit 6 with the standard output file. Also see *setfil* (III) for a way to associate unit numbers with named files.

FILES

a.out	loaded output
f.tmp[123]	temporary (deleted)
/usr/fort/fc1	compiler proper
/lib/fr0.o	runtime startoff
/lib/filib.a	interpreter library
/lib/libf.a	built-in functions, etc.
/lib/liba.a	system library

SEE ALSO

ANSI standard, ld (I) for loader flags. For some subroutines, try ierror, getarg, setfil (III).

DIAGNOSTICS

Compile-time diagnostics are given in English, accompanied if possible with the offending line number and source line with an underscore where the error occurred. Runtime diagnostics are given by number as follows:

1	invalid log argument
2	bad arg count to amod
3	bad arg count to atan2
4	excessive argument to cabs
5	exp too large in cexp
6	bad arg count to cmplx
7	bad arg count to dim
8	excessive argument to exp
9	bad arg count to idim
10	bad arg count to isign
11	bad arg count to mod
12	bad arg count to sign
13	illegal argument to sqrt
14	assigned/computed goto out of range
15	subscript out of range
16	real**real overflow
17	(negative real)**real
100	illegal I/O unit number
101	inconsistent use of I/O unit
102	cannot create output file
103	cannot open input file
104	EOF on input file
105	illegal character in format
106	format does not begin with (
107	no conversion in format but non-empty list
108	excessive parenthesis depth in format
109	illegal format specification
110	illegal character in input field
111	end of format in hollerith specification
112	bad argument to setfil
120	bad argument to ierror
999	unimplemented input conversion

BUGS

The following is a list of those features not yet implemented:
arithmetic statement functions
scale factors on input
Backspace statement.

FILE (I) FILE (I)

NAME

file − determine format of file

SYNOPSIS

file files

DESCRIPTION

File will examine each of its arguments and give a guess as to the contents of the file. It is the only program that will give device numbers of special files.

BUGS

If the file is not instantly recognized, its type is given as 'unknown'. There should be some heuristic to recognize source file 'signatures' in each of the standard languages.

NAME

find − find files

SYNOPSIS

find pathname expression

DESCRIPTION

Find recursively descends the directory hierarchy from *pathname* seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, −*n* means less than *n* and *n* means exactly *n*.

−name filename	True if the *filename* argument matches the current file name. Normal *Shell* argument syntax may be used if escaped (watch out for ‘[’, ‘?’ and ‘*’).
−perm onum	True if the file permission flags exactly match the octal number *onum* (see *chmod* (I)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat* (II)) become significant and the flags are compared: *(flags&onum)==onum.*
−type *c*	True if the type of the file is *c,* where *c* is **b, c, d** or **f** for block special file, character special file, directory or plain file.
−links *n*	True if the file has *n* links.
−user uname	True if the file belongs to the user *uname.*
−size *n*	True if the file is *n* blocks long (512 bytes per block).
−atime *n*	True if the file has been accessed in *n* days.
−mtime *n*	True if the file has been modified in *n* days.
−exec command	True if the executed command returns exit status zero (most commands do). The end of the command is punctuated by an escaped semicolon. A command argument ‘{}’ is replaced by the current pathname.
−ok command	Like **−exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds **y**.
−print	Always true; causes the current pathname to be printed.

The primaries may be combined with these operators (ordered by precedence):

!	prefix *not*
−a	infix *and,* second operand evaluated only if first is true
−o	infix *or,* second operand evaluated only if first is false
(expression)	parentheses for grouping. (Must be escaped.)

To remove files named ‘a.out’ and ‘*.o’ not accessed for a week:

```
find / "(" −name a.out −o −name "*.o" ")" −a −atime +7 −a −exec rm {} ";"
```

FILES

/etc/passwd

SEE ALSO

sh (I), if (I), file system (V)

BUGS

There is no way to check device type.
Syntax should be reconciled with *if.*

NAME

get − get generation from SCCS file

SYNOPSIS

get [−rrel[.lev]] [−ccutoff] [−iincl−list] [−xexcl−list] [−k] [−e] [−l[p]] [−p] [−m] [−n] [−s] name ...

DESCRIPTION

Get generates an ASCII text file from each named SCCS file accoring to the specifications given by its keyletter arguments, which begin with "−". The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files, as determined by their magic numbers (see *sccsfile* (V)), and unreadable files are silently ignored. If a name of "−" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* See **FILES**, below, for an explanation of how the name of this file is determined.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

−r	Release or release and level to be generated. If only the release is specified, the level defaults to the highest level in that release. If the entire argument is omitted, the meaning is the same as if the default release were specified (see *admin* (I)). A level may not be specified if the **e** argument is supplied (see below).
−c	Cutoff date-time. in the form YY[MM[DD[HH[MM[SS]]]]]. No delta which was created after the specified cutoff date-time will be applied. Units omitted from the date-time default to their maximum possible values; that is, "−c7502" means the same as "−c750228235959".
−i	This argument is used to specify a list of deltas to be included (forced to be applied). The list has the following syntax: <list> ::= <range> (<list> , <range> <range>::= <delta> (<delta> − <delta> <delta> ::= <rel> (<rel> . <lev> If a level is omitted from a delta specification the latest level of the specified release is assumed.
−x	This argument is similar to **i** except that it is followed by a list of deltas to be excluded (forced to not be applied).
−k	This argument suppresses re placement of identifiaction keywords (see below) by specific values. The **k** argument is implied by the **i, x** or **e** arguments.
−e	This argument indicates the this *get* is for the purpose of making a delta with a later execution of *delta.* It causes creation of a *p-file* (see **FILES**). Another *get* with an **e** argument may not be executed until the delta is made. If the *g-file* generated by a *get* with an **e** argument is ruined, a new one may be obtained by executing another *get* with a **k**

argumemt instead of an **e**. Note that although the **c**, **i** and/or **x** arguments may be used in combination with **e**, *delta* will not use any of the three when regenerating the *g-file.* When the **e** argument is supplied the protection restrictions determined by the ceiling, the floor, the release locks and the list of users authorized to make deltas are enforced.

−l This argument causes a delta summary to be written into an *l-file* (see **FILES**). if **−lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. The *reform*(VI) command can be used to truncate lines of the *l-file.*

The following procedure can be used to get a neatly formatted *l-file*

```
sed ´s/.*/.x "&"/´ <l-file (nroff m −
```

where file m contains:

```
.ll 80
.pl 1
.na
.in 46
.de x
.ti 0
\\&\\$1
..
```

−p This argument causes the generated text to be written to the standard output instead of to a *g-file.* All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **s** argument is supplied, in which case it disappears.

−s This argument suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

−m This argument caused each generated text line to be preceded by the release and level of the delta which inserted the text line. The format is: a 4-character, blank-filled, right-justified release number, followed by a period, followed by a 4-character, blank-filled, left-justified level number, followed by a horizontal tab, followed by the text line.

−n This argument causes each generated text line to be preceded with the name of the *g-file.* The format is: *g-file* name, followed by a horizontal tab, followed by the text line. When both the **m** and **n** arguments are supplied the format is: *g-file* name, followed by a horizontal tab, followed by the **m** argument format.

For each file processed, *get* responds (on the standard output) with the release and level being accessed and with the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a newline) before it is processed. Unapplied unacknowledged non-propagating deltas are listed along with the notation "Non-prop". If the **i** argument is supplied included deltas are listed following the notation "Included"; if the **x** argument is supplied excluded deltas are listed following the notation "Excluded".

Identifying information is inserted into the generated text by replacing *identification keywords* by appropriate values, wherever they occur. The following keywords are available:

Keyword	*Value*
%M%	Module name (actually, the *g-file* name – see **FILES**).
%R%	Release.
%L%	Level.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%E%	Date of newest applied delta(YY/MM/DD).
%G%	Date of newest applied delta(MM/DD/YY).
%T%	Current time (HH:MM:SS).
%U%	Time of newest applied delta (HH:MM:SS).
%X%	"P" if any unacknowledged non-propagating deltas were not applied (see *delts*(I)); otherwise blank.
%Z%	The 4 characters @(#) (used to construct strings recognizable by *what*(I).
%P%	The person field of the header (see *admin*(I)).
%Y%	The type field of the header (see *admin*(I)).
%S%	The description field of the header (see *admin*(I)). RJE users N.B.: the description can be up to 99 (i.e. greater than 80) characters long.

FILES

Several auxiliary files may be created by *get.* These files are known generically as the *g-file, l-file,* and *p-file.* The letter before the hyphen is called the tag. An auxiliary file name is formed fron the SCCS file name: all SCCS file names must be of the form "**s.***modulename*", the auxiliary files are named by replacing the leading "s" with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the "s". For example, if the SCCS file name is "s.xyz.c", the auxiliary file names would be "xyz.c", "l.xyz.c" and "p.xyz.c" respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **p** argument is supplied, or zero lines of text were generated). It is owned by the real user. If the **k** argument is supplied or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission (either explicitly or implicitly) in the current directory.

The *l-file* is also created (unless a **p** follows the **−l**) in the current directory, if the **l** argument is supplied; its mode is 444 and it is owned by the real user. Only the real user need nave write permission (either explicitly or implicitly) in the current directory. The *l-file* contains a table showing which deltas were applied. The table has the following fixed format:

Columns	Contents
1	Blank if delta was applied; "*" otherwise.
2	A code indicating a "special" reason why the delta was or was not applied: "I": Included. "X": Excluded. "C": Cut off (by a **c** argument). "P": Non-propagating.
3-6	Release number (right justified).
7	"."
8-11	Level number (left justified).

12-35	Date and time (in the form returned by *ctime*(III) of creation.)
37-42	Creator (left justified).
44-?	History (maximum of 200 characters).

The *p-file* is used to pass information resulting from a *get* with an e argument along to *delta.* Its existence also serves to prevent a subsequent execution of *get* with an e argument until the *delta* is executed. The *p-file* is created in the directory containing the SCCS file (which might, of course, also be the current directory), and the effective user must have write permission (either explicitly or implicitly) in that directory. Its mode is 444 and it is owned by the effective user. The format of the *p-file* is: the release number (in ASCII), followed by a blank, followed by the login name of the real user, followed by a newline.

SEE ALSO

admin(I), delta(I), prt(I), what(I), help(I), ctime(III), sccsfile(V),
SCCS/PWB User's Manual, The Source Code Control System (which should not be taken too literally).

DIAGNOSTICS

Use *help*(I) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the e argument is supplied.

GOTO (I) GOTO (I)

NAME

goto – command transfer

SYNOPSIS

goto label

DESCRIPTION

Goto is allowed only when the Shell is taking commands from a file. The file is searched from the beginning for a line beginning with ':' followed by one or more spaces followed by the *label.* If such a line is found, the *goto* command returns. Since the read pointer in the command file points to the line after the label, the effect is to cause the Shell to transfer to the labelled line.

Note that to the Shell, any line beginning with a ':' is a comment.

SEE ALSO

sh (I)

BUGS

GREP(I) GREP(I)

NAME

grep − search a file for a pattern

SYNOPSIS

grep [**−v**] [**−b**] [**−c**] [**−n**] expression [file] ...

DESCRIPTION

Grep searches the input files (standard input default) for lines matching the regular expression. Normally, each line found is copied to the standard output. If the **−v** flag is used, all lines but those matching are printed. If the **−c** flag is used, only a count of matching lines is printed. If the **−n** flag is used, each line is preceded by its relative line number in the file. If the **−b** flag is used, each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

In all cases the file name is shown if there is more than one input file.

For a complete description of the regular expression, see ed (I). Care should be taken when using the characters $ * [^ | ( ) and \ in the regular expression as they are also meaningful to the Shell. It is generally necessary to enclose the entire *expression* argument in quotes.

SEE ALSO

ed (I), sh (I)

BUGS

Lines are limited to 256 characters; longer lines are truncated.

NAME

help — ask for help

SYNOPSIS

help [arg]

DESCRIPTION

Help finds information to explain a message from an SCCS command or the use of an SCCS command. The argument is either a message number or a command name. Message numbers normally appear in parentheses after the message. The program responds with explanatory information.

If the argument is omited, the program will prompt for the argument.

When all else fails, try "help stuck".

FILES

The ASCII file /usr/lib/sccs.hf is used to store explanations. Anyone wishing to modify the file should list out a little of it — the format will be obvious.

DIAGNOSTICS

Use *help* for them.

NAME

if − conditional command

SYNOPSIS

if expr command [arg ...]

DESCRIPTION

If evaluates the expression *expr,* and if its value is true, executes the given *command* with the given arguments.

The following primitives are used to construct the *expr:*

−**r** file	true if the file exists and is readable.
−**w** file	true if the file exists and is writable.
s1 = s2	true if the strings *s1* and *s2* are equal.
s1 **!=** s2	true if the strings *s1* and *s2* are not equal.
{ command } [n]	The bracketed command is executed to obtain the exit status. Status zero is considered *true.* The command must not be another *if.* The optional *n* is a valid condition code (0 = success, 4 = partial failure, 8 = total failure) When used, status <= n is considered to be *true.*

These primaries may be combined with the following operators:

!	unary negation operator
−**a**	binary *and* operator
−**o**	binary *or* operator
(expr)	parentheses for grouping.

−**a** has higher precedence than −**o.** Notice that all the operators and flags are separate arguments to *if* and hence must be surrounded by spaces. Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO

sh (I), exit (II)

BUGS

Does not work with special commands (e.g. shift,read...) whose code is embedded in the shell.

KILL(I) KILL(I)

NAME

kill − terminate a process

SYNOPSIS

kill [−sig] processid ...

DESCRIPTION

Kills the specified processes. The process number of each asynchronous process started with '&' is reported by the Shell. Process numbers can also be found by using *ps* (I).

If process number 0 is used, then all processes belonging to the current user and associated with the same process group are killed.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by "−" is given as first argument, that signal is sent instead of catchable *kill* (see *signal (II)).*

SEE ALSO

ps (I), sh (I), signal (II)

BUGS

NAME

lc − LIL compiler

SYNOPSIS

lc [−c] [−p] [−P] file ...

DESCRIPTION

Lc is the UNIX LIL compiler. It accepts three types of arguments:

Arguments whose names end with '.l' are taken to be LIL source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.l'. The '.o' file is normally deleted, however, if a single LIL program is compiled and loaded all at one go.

The following flags are interpreted by *lc.* See *ld* (I) for load-time flags.

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls the *monitor* (III) subroutine at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof* (I).

−P Run only the macro preprocessor on the named LIL programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be either loader flag arguments, or LIL-compatible object programs, typically produced by an earlier *lc* run, or perhaps libraries of LIL-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

FILES

file.l	input file
file.o	object file
a.out	loaded output
/tmp/ctm0?	temporary
/usr/lib/lil[12]	compiler
/lib/liba.a	Assembler library

SEE ALSO

monitor (III), prof (I), ld (I).

DIAGNOSTICS

The diagnostics produced by LIL itself are intended to be self-explanatory.

BUGS

NAME

ld — link editor

SYNOPSIS

ld [−sulxXSrdnioeb] name ...

DESCRIPTION

Ld combines several object programs into one; resolves external references; and searches libraries. In the simplest case the names of several object programs are given, and *ld* combines them, producing an object module which can be either executed or become the input for a further *ld* run. (In the latter case, the **−r** option must be given to preserve the relocation bits.) The output of *ld* is left on **a.out.** This file is made executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

Ld understands several flag arguments which are written preceded by a '−'. Except for **−l**, they should appear before the file names.

−s	'strip' the output; that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip.*
−u	take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
−l	This option is an abbreviation for a library name. **−l** alone stands for '/lib/liba.a', which is the standard system library for assembly language programs. **−l***x* stands for '/lib/lib*x*.a' where *x* is a character string. If the file is nonexistent in /lib, then /usr/lib is searched for the corresponding file. A library is searched when its name is encountered, so the placement of a **−l** is significant.
−x	do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.
−X	Save local symbols except for those whose names begin with 'L'. This option is used by *cc* to discard internally generated labels while retaining symbols local to routines.
−S	Discard all symbols except locals and globals.
−r	generate relocation bits in the output file so that it can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols, and suppresses the 'undefined symbol' diagnostics.
−d	force definition of common storage even if the **−r** flag is present.
−n	Arrange that when the output file is executed, the text (program) portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 4K word boundary following the end of the text.

−i	When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and **−n** is that here the data starts at location 0.
−o	Take the following argument as the output file rather than **a.out.** A previously existing **a.out** file will not be altered.
−e	Take the following argument as a symbol and use it as the entry point on execution. The default entry point is location zero of the text segment, or the first defined entry point from the input files.
−b	Take the following argument as a decimal (octal if preceeded by 0) number to be used as the virtual text origin. All relocatable references will be based from this number rather than zero. This is currently useful only for foreign execution.

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	libraries
a.out	output file

SEE ALSO

as (I), ar (I), strip (I)

DIAGNOSTICS

BUGS

NAME

line — get terminal name

SYNOPSIS

line

DESCRIPTION

Line gives the name of the user's terminal in the form 'ln*xx*' where *xx* a single or double character(s). The actual path name is then '/dev/ln*xx*'.

DIAGNOSTICS

'nonexistent line' if the standard input file is not a terminal.

BUGS

LN (I) LN (I)

NAME

ln — make a link

SYNOPSIS

ln name1 [name2]

DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

Ln creates a link to an existing file *name1*. If *name2* is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of *name1*.

It is forbidden to link to a directory or to link across file systems.

SEE ALSO

rm (I)

BUGS

Tp doesn't understand about links and makes one copy for each name by which a file is known; thus if the tape is extracted, several copies are restored and the information that links were involved is lost.
If *name2* is a directory, *ln* should make a link to the file *name1* in that directory with the name *name1*.

LOGIN(I) LOGIN(I)

NAME

login — sign onto UNIX

SYNOPSIS

login [username]

DESCRIPTION

The *login* command is used when a user initially signs onto UNIX, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See 'How to Get Started' for how to dial up initially.

If *login* is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

After a successful login, accounting files are updated and the user is informed of the existence of *.mail.* If the message-of-the-day file, */etc/motd,* exists, it is printed on the user's terminal. *Login* initializes the user and group IDs and the working directory, then executes a command interpreter (usually *sh* (I)) according to specifications found in a password file.

Login is recognized by the Shell and executed directly (without forking).

FILES

/etc/utmp	instantaneous accounting
/etc/wtmp	historical accounting
.mail	mail
/etc/motd	message-of-the-day
/etc/passwd	password file

SEE ALSO

init (VIII), getty (VIII), mail (I), passwd (I), passwd (V)

DIAGNOSTICS

'login incorrect,' if the name or the password is bad. 'No Shell,', 'cannot open password file,' 'no directory': consult a UNIX programming counselor.

BUGS

NAME

lpr − line printer spooler

SYNOPSIS

lpr [−] [**+**] [**+**−]file ...

DESCRIPTION

Lpr arranges to have the line printer daemon print the file arguments.

Normally, each file is printed in the state it is found when the line printer daemon reads it. Files following a **+** will be copied before printing. Files following a − will be unlinked (removed) by *lpr* after printing.

If there are no arguments, then the standard input is read and on-line printed. Thus *lpr* may be used as a filter.

FILES

/usr/lpd/*	spool area
/etc/passwd	personal ident cards
/etc/lpddaemon	

SEE ALSO

lpd (VIII), passwd (V)

BUGS

NAME

ls — list contents of directory

SYNOPSIS

ls [**−ltasdruif**] name ...

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. There are several options:

−l list in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file, the size field will contain the major and minor device numbers instead.

−t sort by time modified (latest first) instead of by name, as is normal

−a list all entries; usually those beginning with '.' are suppressed

−s give size in blocks for each entry

−d if argument is a directory, list only its name, not its contents (mostly used with **−l** to get status on directory)

−r reverse the order of sort to get reverse alphabetic or oldest first as appropriate

−u use time of last access instead of last modification for sorting (**−t**) or printing (**−l**)

−i print i-number in first column of the report for each file listed

−f force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **−l**, **−t**, **−s**, and **−r**, and turns on **−a**; the order is the order in which entries appear in the directory.

The mode printed under the **−l** option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
− if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions, the next to permissions to others in the same user-group, and the last to all others. Within each set the three characters indicate permission to read, to write, or to execute the file as a program, respectively. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
− if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise, the user-execute permission character is given as **s** if the file has set-user-ID mode.

FILES

/etc/passwd to get user ID's for **ls −l**.

BUGS

MAIL(I) MAIL(I)

NAME

mail − send mail to designated users

SYNOPSIS

mail [**−yn**] [person ...]

DESCRIPTION

Mail with no argument searches for a file called *.mail*, prints it if it is nonempty, then asks if it should be saved. If the answer is y, the mail is added to *mbox*. Finally *.mail* is truncated to zero length. To leave *.mail*, hit 'delete.' The question can be answered on the command line with the argument '−y' or '−n'.

When **persons** are named, *mail* takes the standard input up to an end of file and adds it to each *person's .mail* file. The message is preceded by the sender's name and a postmark.

A *person* is either a user name recognized by *login* (I), in which case the mail is sent to the initial working directory of that user; or the path name of a directory, in which case *.mail* in that directory is used.

When a user logs in he is informed of the presence of mail. No mail will be received from a sender to whom *.mail* is inaccessible or unwritable.

FILES

/etc/passwd	to identify sender and locate persons
/etc/utmp	to identify sender
.mail	input mail
mbox	saved mail
/tmp/m*	temp file

SEE ALSO

write (I)

BUGS

When it is impossible to send mail, the sender is not always notified of the failure, i.e. when the "receiver" directory is protected and no ".mail" exists.

MESG(I) MESG(I)

NAME

mesg — permit or deny messages

SYNOPSIS

mesg [n] [y]

DESCRIPTION

Mesg with argument **n** forbids messages via *write* by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reverses the current permission. In all cases the previous state is reported.

FILES

/dev/ln??

SEE ALSO

write (I)

DIAGNOSTICS

'?' if the standard input file is not a terminal.

BUGS

NAME

mkdir — make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. The standard entries '.' and '..' are made automatically.

SEE ALSO

rmdir (I)

BUGS

NAME

mtm − magnetic tape manipulation

SYNOPSIS

mtm [−**s***n*] [−**l***m*] [−**b***p*] [*unit*]

DESCRIPTION

Mtm helps in the processing of multifile magnetic tapes. The optional arguments are:

−**s***n*	Forward space the magnetic tape for *n* files.
−**l***m*	Produce a list of the number and sizes of the records on the magnetic tape for *m* files. If *m* is missing *mtm* will analyze records up to a double EOT.
−**b***p*	Define the maximum record size as *p*K bytes. If *b* is missing, then the maximum record size is assumed to be 2K bytes.
unit	*Unit* specifies the drive on which the tape is mounted. If *unit* is missing, drive 0 is assumed.

As an example,

```
mtm −s1 −l2 1
```

gives:

```
File 2:
          Record 1 − 14 bytes
          Record 2 − 512 bytes
          66 Records
File 3:
          Record 1 − 512 bytes
          68 Records
DONE
```

This means that file 2 contains one 14-byte record and two 512-byte records.

The two arguments can be combined to skip some files, then list some number of remaining files. If no arguments are given, *mtm* will list the entire tape.

FILES

/dev/rmt?

SEE ALSO

tm (IV), ht (IV)

BUGS

This command should be replaced by an stty for tape motion.

NAME

mv − move or rename a file

SYNOPSIS

mv [−f] name1 name2

DESCRIPTION

Mv changes the name of a file from *name1* to *name2.* If *name2* is a directory, *name1* is moved to that directory with its original file-name. Directories may only be moved within the same parent directory (just renamed).

If *name2* already exists, it is removed before *name1* is renamed. If *name2* has a mode which forbids writing, *mv* prints the mode and reads the standard input to obtain a line; if the line begins with **y**, the move takes place; if not, *mv* exits. The optional argument **-f** allows the move to take place without verification.

If *name2* would lie on a different file system, so that a simple rename is impossible, *mv* copies the file and deletes the original.

BUGS

MVALL(I) MVALL(I)

NAME

mvall − move all files to a directory

SYNOPSIS

mvall name [...] dir

DESCRIPTION

The named files are moved into directory *dir* by means of successive calls to the command *mv(I).* These calls are of the form

```
mv name dir/.
```

Only ordinary files are moved by *mvall.* Directories and special files are skipped, and a comment is printed.

SEE ALSO

mv(I)

NAME

neqn − typeset mathematics on terminal

SYNOPSIS

neqn [file] ...

DESCRIPTION

Neqn is an nroff (I) preprocessor. The input language is the same as that of eqn (I). Normal usage is almost always

```
neqn file ... | nroff
```

Output is meant for terminals with forward and reverse capabilities, such as the Model 37 teletype or GSI terminals.

If no arguments are specified, *neqn* reads the standard input, so it may be used as a filter.

SEE ALSO

eqn (I), gsi (VI)

BUGS

Because of some interactions with *nroff* there may not always be enough space left before and after lines containing equations.

NICE (I) NICE (I)

NAME

nice − run a command at low priority

SYNOPSIS

nice command [arguments]

DESCRIPTION

Nice executes *command* at low priority.

SEE ALSO

nohup(I), nice(II)

BUGS

NM(I) NM(I)

NAME

nm − print name list

SYNOPSIS

nm [**−cnrupg**] [name]

DESCRIPTION

Nm prints the symbol table from the output file of an assembler or loader run. Each symbol name is preceded by its value (blanks if undefined) and one of the letters **U** (undefined) **A** (absolute) **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), or **C** (common symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

If no file is given, the symbols in **a.out** are listed.

Options are:

−c	List only C-style external symbols, that is those beginning with underscore '_'.
−g	Print only global (external) symbols.
−n	Sort by value instead of by name.
+p **−p**	Don't sort; print in symbol-table order.
−r	Sort in reverse order.
−u	Print only undefined symbols.

FILES

a.out

BUGS

Trouble with files greater than 32K words.

NOHUP (I)

NAME

nohup − run a command immune to hangups

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups, quits and interrupts all ignored.

SEE ALSO

nice (I), signal (II)

BUGS

Beware of *nohup* on command lines with pipes.

NAME

nroff – Format text for typewriter-like devices

SYNOPSIS

nroff [*options*] *files*

DESCRIPTION

NROFF accepts lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
–**o***list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N–M* and means pages *N* through *M;* an initial *–N* means from the beginning to page *N;* and a final *N–* means from *N* to the end.
–**n***N*	Number first generated page *N*.
–**s***N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline.
–**m***name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files.*
–**r***aN*	Register *a* (one-character) is set to *N.*
–**i**	Read standard input after the input files are exhausted.
–**q**	Invoke the simultaneous input-output mode of the **rd** request.
–**T***name*	Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 teletype, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).
–**e**	Produce equally-spaced words in adjusted lines, using full terminal resolution.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta00000	temporary file
/usr/lib/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables

SEE ALSO

NROFF/TROFF User's Manual (internal memorandum).
tbl(I), neqn(I)

OD(I) OD(I)

NAME

od − octal dump

SYNOPSIS

od [**−abcdho**] [file] [[**+**] offset[**.**][**b**]]

DESCRIPTION

Od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing **−o** is default. The meanings of the format argument characters are:

a	interprets words as PDP-11 instructions and dis-assembles the operation code. Unknown operation codes print as ???.
b	interprets bytes in octal.
c	interprets bytes in ascii. Unknown ascii characters are printed as \?.
d	interprets words in decimal.
h	interprets words in hex.
o	interprets words in octal.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used. Thus *od* can be used as a filter.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If '**.**' is appended, the offset is interpreted in decimal. If '**b**' is appended, the offset is interpreted in blocks. (A block is 512 bytes.) If the file argument is omitted, the offset argument must be preceded by '**+**'.

Dumping continues until end-of-file.

SEE ALSO

adb (I)

ONINTR (I) ONINTR (I)

NAME

onintr − specify interrupt processing for a command file

SYNOPSIS

onintr [label]

DESCRIPTION

Onintr specifies that an interrupt processing routine begins at : 'label'. The interrupt processing continues until a *return,* an *exit,* or the end of the command file is encountered. If an interrupt occurs on a command line **n,** control passes to the interrupt routine, and upon completion returns to the command line **n+1,** unless the completion was caused by an *exit* command, whereupon *sh* terminates the command file.

Onintr used without an argument specifies that upon interrupt the command file will terminate. During the execution of an interrupt process routine, interrupts are suppressed.

SEE ALSO

sh (I), return (I), exit (I), goto (I)

FILES

DIAGNOSTICS

BUGS

NAME

passwd – change login password

SYNOPSIS

passwd name password

DESCRIPTION

The *password* becomes associated with the given login name. This can only be done by corresponding user or by the super-user. An explicit null argument ("") for the password argument removes any password.

FILES

/etc/passwd

SEE ALSO

login (I), passwd (V), crypt (III)

BUGS

NAME

pfe — print floating exception

SYNOPSIS

pfe

DESCRIPTION

Pfe examines the floating point exception register and prints a diagnostic for the last floating point exception.

SEE ALSO

signal (II)

BUGS

Since the system does not save the exception register in a core image file, the message refers to the last error encountered by anyone. Floating exceptions are therefore volatile.

PR(I) PR(I)

NAME

pr – print file

SYNOPSIS

pr [**–h** *header*] [–*n*] [**+***n*] [**–w***n*] [**–l***n*] [**–t**] [**–s***c*] [**–m**] name . . .

DESCRIPTION

Pr produces a printed listing of one or more files. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, *pr* prints its standard input, and is thus usable as a filter.

Options apply to all following files but may be reset between files:

–*n*	produce *n*-column output
+*n*	begin printing with page *n*
–h	treat the next argument as a header to be used instead of the file name
–w*n*	for purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72
–l*n*	take the length of the page to be *n* lines instead of the default 66
–t	do not print the 5-line header or the 5-line trailer normally supplied for each page
–s*c*	separate columns by the single character *c* instead of by the appropriate amount of white space. A missing *c* is taken to be a tab.
–m	print all files simultaneously, each in one column

Interconsole messages via *write* (I) sent by other than super user are forbidden during a *pr*.

FILES

/dev/ln?? to suspend messages.

SEE ALSO

cat (I), cp (I)

DIAGNOSTICS

none; files not found are ignored

BUGS

Many, will not always print files of (0-10) length.

PROF (I)

NAME

prof − display profile data

SYNOPSIS

prof [**−a**] [**−l**] [file]

DESCRIPTION

Prof interprets the file *mon.out* produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file (*a.out* default) is read and correlated with the *mon.out* profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **−a** option is used, all symbols are reported rather than just external symbols. If the **−l** option is used, the output is listed by symbol value rather than decreasing percentage.

In order for the number of calls to a routine to be tallied, the **−p** option of *cc* must have been given when the file containing the routine was compiled. This option also arranges for the *mon.out* file to be produced automatically.

FILES

mon.out	for profile
a.out	for namelist

SEE ALSO

monitor (III), profil (II), cc (I)

BUGS

Beware of quantization errors.

NAME

prt − print SCCS file

SYNOPSIS

prt [**−hrdba**] [**−y**[rel[.lev]]] [**−c**[cutoff]] [**−psx**] name ...

DESCRIPTION

Prt prints part or all of an SCCS file in a useful format. If no **h, r, b,** or **d** keyletter arguments are supplied, the **d** keyletter is assumed. If a directory is named, *prt* behaves as though each file in the directory were specified as a named file, except that non-SCCS files, as determined by their magic numbers (see *sccsfile* (V)), and unreadable files are silently ignored. If a name of "−" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored.

One or a combination of the arguments **h, r, d,** and **b** cause *prt* to print the header, release table, delta table, and body respectively. If one of the **h, r,** or **b** keyletters is supplied, or none of the **y** or **c** keyletters are supplied, the name of the file being processed (preceded by one newline and followed by two newlines) is printed.

A delta table entry normally has the following fixed format:

Columns	*Contents*
1	Type of delta: "D" ordinary, propagating "U" unacknowledged, non−propagating "P" acknowledged, non−propagating "R" removed "H" changed history "C" combined
2−5	Release number (right justified).
6	"."
7−10	Level number (left justified)
11−34	Date and time (in the form returned by *ctime*(III)) of creation.
36−41	Creator (left justified).
43−?	History (maximum of 200 characters).

The **y** keyletter is used to cause the printing of the delta table to stop with the printing of a specific delta. A release and level may be supplied with the **y** keyletter. The default release (see *admin*(I)) is used if none is specified. The highest level of the specified release is used if no level is specified. If the named delta does not exist, the entire delta table is printed. When the **y** keyletter is specified, each line of the delta table is preceded by the name of the SCCS file being processed, followed by a ":", followed by a horizontal tab.

The **c** keyletter is used to cause the printing of the delta table to stop when the entry about to be printed is older than the specified date-time, or until all entries have been printed. A cutoff date-time (see *get*(I)) may be supplied with the **c** keyletter. If no date-time is supplied, the epoch 0000 GMT Jan. 1, 1970 is used. As with the **y** keyletter, when the **c** keyletter is specified, each line of the delta table is preceded by the file name of the SCCS file being processed, followed by a ":", followed by a horizontal tab. The delta table, however, is printed in a different format when the **c** keyletter is supplied:

Columns	*Contents*
1−8	Date of creation (in the form YY/MM/DD).
9	blank
10−17	Time of creation (in the form HH:MM:SS).
18	Horizontal tab.
19	Type of delta (see above).
20−23	Release number (right justified).
24	"."
25−28	Level number (left justified).
29−34	Creator (left justified).
35−?	History (maximum of 200 characters).

This form of the delta table makes it easy to sort multiple delta tables by time order. For example, the following will print the delta tables of all SCCS files in directory *sccs* in reverse chronological order:

```
prt −c sccs | sort ′−rthorizontal−tab′ +1 −2
```

When both the **y** and **c** keyletters are supplied, *prt* will stop printing when the first of the two conditions is met.

The *reform*(I) command can be used to truncate long lines. *Nroff*(I) can be used to get a neatly formatted delta table; e.g.:

```
prt sccs | sed ′s/.*/.x "&"/′ | nroff m −
```

where file m contains:

```
.ll 80
.pl 1
.na
.in 46
.de x
.ti 0
\\&\\$1
..
```

The USG distribution of *prt* implements the **n** value option to the **−d** keyletter as per the SCCS/PWB User's Guide. A **−dn** causes the proper *nroff*(I) formatting requests to be prepended to the output. A number may follow **−dn** to specify line length, 80 is the default.

If the **a** keyletter is not specified then "R" (removed), "H" (changed history), and "C" (combined) delta entries are not printed.

None of the **c, y,** or **a** keyletters have any effect unless the **d** keyletter is also specified (or assumed).

The **p, s,** and **x** keyletters are reserved for SCCS support personnel (those who install and/or maintain SCCS) use only. They should not be used by anyone else.

See *admin*(I), *sccsfile*(V), and *SCCS/PWB User's Manual* for more information about the meaning of the output of *prt.*

SEE ALSO

get(I), delta(I), prt(I), what(I), help(I), sccsfile(V)
SCCS/PWB User's Manual

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

ps – process status

SYNOPSIS

ps [-alstx] [ttyname] [file] [namelist]

DESCRIPTION

Ps prints certain data about active processes. If a file is specified, it must contain the name of an alternate core image. If namelist is specified (for use with the **-s** option), it must be a replacement for /unix. There are several options:

a asks for information about all processes associated with terminals. Ordinarily, only one's own processes are displayed.

s uses the last argument as the namelist in place of /unix.

t indicates that the next argument is the name of the terminal desired. This may be a one or two-character name.

x reports on all processes not associated with any terminals.

l provides a long listing which contains the following meaningful columns:

Heading	Description
F	a number encoding the flags associated with a process. It may be any combination of the following: 00 swapped-out process 01 swapped-in process 02 the scheduler 04 process locked in core 30 tracing
S	a letter encoding the state of the process: S - sleeping W - waiting R - ready to run I - idling (unused) Z - zombie - process exited, parent not yet notified. T - traced
UID	the real user-id of the owner of the process.
PID	the unique process number; this is useful with *kill*.
PPID	the id of the parent process.
CPU	a magic number related to the amount of process time received by that process.
PRI	the priority of the process; high numbers mean low priority

NICE	priority penalty received by a process.
ADDR	the location of the process; in 64-byte granularity for swapped-in processes; in 512-byte granularity for swapped-out processes.
SZ	the size in blocks of the core image of the process.
WCHAN	the core address in the system of the event for which the process is waiting. If blank, the process is running.
LNE	the last character(s) of the process' controlling terminal.
TIME	total cpu time, system and user (in minutes:seconds), consumed by the process.
COMMAND	the file name of the process.

Ps attempts to learn the file name and arguments given when the process was created, by examining core memory and/or the swap area.

FILES

/unix	system namelist
/dev/mem	core memory
/dev/null	swap device with alternate core image file

SEE ALSO

kill(I)

BUGS

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

chdir (I)

BUGS

READ (I) READ (I)

NAME

read, open, onend – sequential file read

SYNOPSIS

onend [label]
open [fname]
read

DESCRIPTION

Within a command file, read accesses data, one line at a time from another file. The *read* is done sequentially from the beginning of the file.
Open is needed to identify the file to be read. Only one file may be open at any time. Without an argument the standard input is used.
Onend provides a branch to *label* when the EOF condition is reached on *fname.* If *label* is not supplied, a diagnostic is written to the error output,and a null string is sent to the standard output.

FILES

SEE ALSO

sh (I), goto (I), read (II)

DIAGNOSTICS

BUGS

Standard Shell syntax is violated to allow pipe and redirected input to take precedence over an *open* file, which in turn takes precedence over standard Shell input.
Don't try to read a directory file.

NAME

return – terminate profile or interrupt processing routine

SYNOPSIS

return [rtncode]

DESCRIPTION

Return terminates processing of a profile file or an interrupt processing routine within a command file. In any other case, *sh* treats *return* as an exit.

The return code is set to the optional argument 'rtncode', otherwise to 0.

SEE ALSO

sh(I), exit(I)

REW(I) REW(I)

NAME

rew — rewind tape

SYNOPSIS

rew [[**m**]digit]

DESCRIPTION

Rew rewinds DECtape or magtape drives. The digit is the logical tape number, and should range from 0 to 7. if the digit is preceded by **m**, *rew* applies to magtape rather than DECtape. A missing digit indicates drive 0.

FILES

/dev/tap?
/dev/mt?

BUGS

Magnetic tape units 4-7 map into physical drives 0-3.

RM (I) **RM (I)**

NAME

rm – remove (unlink) files

SYNOPSIS

rm [−f] [−r] name ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission, *rm* prints the file name and its mode, then reads a line from the standard input. If the line begins with **y**, the file is removed, otherwise it is not. The file is removed without the question being asked if option **−f** was given or if the standard input is not a typewriter.

If a designated file is a directory, an error comment is printed unless the optional argument **−r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory. To remove directories *per se* see rmdir(I).

FILES

/etc/glob to implement the **−r** flag

SEE ALSO

rmdir (I)

BUGS

When *rm* removes the contents of a directory under the **−r** flag, full pathnames are not printed in diagnostics.

RMDIR (I) RMDIR (I)

NAME

rmdir — remove directory

SYNOPSIS

rmdir dir ...

DESCRIPTION

Rmdir removes (deletes) directories. The directory must be empty (except for the standard entries '.' and '..', which *rmdir* itself removes).

BUGS

Needs a **−r** flag.
Write permission in the directory's parent should be required, but is *not*.
Mildly unpleasant consequences can follow removal of your own or someone else's current directory.

NAME

sed − stream editor

SYNOPSIS

sed [**−g**] [**−n**] [**−f** commandfile] ... [[**−e**] command] ... [file] ...

DESCRIPTION

Sed copies the input *files* (standard input default) to the standard output, perhaps performing one or more editor commands (see *ed* (I)) on each line.

The following flags are interpreted by *sed* :

−e Indicates that the next argument is an editor command.

−f Indicates that the next argument is a filename; the file contains editor commands, one to a line. Commands which are inherently multi-line, like *a* or *c,* should be written with the interior newlines preceded by '\'. Append mode is terminated by an unhidden newline.

The **−e** and **−f** flags may be intermixed in any order. If no **−e** or **−f** flags are given, the first argument is taken by default to be an editor command.

Addresses are allowed. The meaning of two addresses is: "Attempt this command on the first line matching the first address, and on all subsequent lines until the next line containing a match of the second address; then begin watching for a match of the first address and iterate." One address means: "Attempt this command on all the lines which match the address." Either line-numbers or regular expressions are allowable as addresses. A line number matches the cumulative line number of the input files. A '$' as an address matches the last line of the last input file.

−g Indicates that all *s* commands should be executed as though followed by a *g.* If only some substitutes are to be done globally, leave out the **−g** flag, and put the *g*'s at the end of the appropriate command lines.

−n Prevents the copying of lines to the output by default. Only lines which are explicitly printed by *p* commands are written. In order to avoid getting double copies of some lines, the *p* command is a no-op unless the **−n** flag is set.

The intention is to simulate the editor as exactly as possible, but the line-at-a-time operation makes certain differences unavoidable or desirable:

1. There is no notion of '.'; and no relative addressing.

2. Commands with no addresses are defaulted to *1,$* rather than to dot.

3. Context addresses must be delimited by '/'; '?' is an error.

4. Expressions in addresses are not allowed.

5. Commands may have only as many addresses as they can use. That is, no command may have more than two addresses; the *a, i, r,* and *l* commands may have only one address.

6. A *p* at the end of a command only works with the *s* command. For other commands, or if the **−n** flag is not in effect, a *p* at the end of a command line is a no-op.

7. A *w* may appear at the end of a substitute command. It should be followed by a single space and a file name. If the substitute command is successfully executed, the line is written to the file. All files are opened when the commands are being compiled, and closed when the program terminates. Only ten different file names may appear in *w* commands in a single run. Unlike *p, w* takes effect regardless of the **−n** flag. If both *p* and *w* are appended to the same substitute command, they must be in the order *pw.*

8. The only commands available are *a, c, d, i, s, p, q, r, w, g, v,* and =. A successful execution of a *q* command causes the current line to be written out if it should be, and execution terminated. When a line is deleted by a *d* or *c* command, no further commands are attempted on the corpse, but another line is immediately read from the input (but see the next point (9)).

9. If an *a, i,* or *r* command is successfully executed, the text is inserted into the file whether or not the line on which the match was made is later deleted or not. Thus the commands:

   ```
   /b/a\
   XXX
   /b/,/c/d
   ```

   applied to the file

   ```
   a
   b
   c
   d
   ```

   will produce

   ```
   a
   XXX
   d
   ```

   on the output.

10. Text inserted in the output stream by the *a, i, c* or *r* commands is not scanned for any pattern matches, nor are any editor commands applied to it.

Syntax:

Blank lines, blanks, and tabs at the beginning of a line in the command file are completely ignored.

Commands may be grouped by curly braces. The opening brace must appear in the place where a function would ordinarily appear; the closing brace must appear on a line by itself (except, of course, for leading blanks or tabs).

If the first line of a command file has #n as its first two characters, the no-copy flag is set, as though the **−n** option had been given on the command line. The rest of the line is ignored; it may be used for a title or a comment.

Pattern Match Commands:

These three capital letter commands are intended to allow pattern matches across new-line characters in the input file.

N Next line is appended to the current line; the two lines are separated by a new-line character which may be matched by '\n' (see below).

D Delete up to and including first (leftmost) new−line in the current pattern space. If the pattern space becomes empty (the only new-line was at the end of the space), read another line from the input. In any case, begin the list of editing commands again from the beginning.

P Print up to and including the first new−line in the pattern space.

Meta−Characters:

\n	Matches imbedded newlines in the pattern space.
\\	Matches '\'
\]	Matches ']'

Examples:

The file /mnt/btl/dir contains telephone-book entries. Most are on a single line; double line entries are signaled by having the second line begin with a blank.

To print out all double-line entries:

```
sed −n
        N
        /\n/p
        D /mnt/btl/dir
```

To combine all double line entries into single lines:

```
sed −n
        N
        /\n/{
                s// /p
                j (see flow−of−control commands)
        }
        P
        D /mnt/btl/dir >newdir
```

Next Line Command:

n The current line is replaced by the next line from the input file, and the list of editing functions is continued after the *n* function.

Flow−of−control Commands:

These commands do no editing on the input line, but serve to control the order in which multipule editing commands are applied to an input line.

:<label>

The label command marks a place in the list of editing commands which may be referred to by *j* and *t* commands. The <label> may be any sequence of eight or fewer characters; if two different colon commands have identical labels, a compile time diagnostic will be generated, and no execution attempted.

j <label>

The jump command causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon command with the same <label> was encountered. If no colon command with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is at tempted.

A *j* command with no <label> is taken to be a jump to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

t <label>

The *t* command tests whether *any* successsful substitutions have been made on the current input line; if so, it jumps to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, and

2) executing a *t* command.

Example:

```
#n      This file prints each non-blank line following a blank line.
1{
        /./p
}
/ $/{
: loop
        n
        /./{
                p
                j
        }
        j loop
}
```

FILES

SEE ALSO

ed (I)

DIAGNOSTICS

BUGS

Lines are silently truncated at 512 characters.

NAME

sh – shell (command interpreter)

SYNOPSIS

sh [**–v**] [**–i**] [**–tc**] [name [arg1 ... [arg9]]]

DESCRIPTION

Sh is the standard command interpreter. It reads and arranges the execution of the command lines typed by the user. Unless an alternative is specified in the entry of a user in */etc/passwd,* a copy of *sh* is provided at login time. *Sh* may itself be called as a command to interpret command lines and/or command files. Before discussing the arguments to the Shell used as a command, the structure of command lines themselves will be given.

Commands. Each command is a sequence of arguments separated by blanks or tabs. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument is the name of an executable file, it is invoked; otherwise the string '/bin/' is prefixed to the argument. (In this way most often-used commands, which reside in '/bin', are found.) If no such command is found, the string '/usr' is further prefixed (to give '/usr/bin/command') and another attempt is made to execute the resulting file name. (Certain lesser-used commands live in '/usr/bin'.) If '/usr/bin/command' exists with executable mode, but does not have the form of an executable program, it is used by *sh* as a file of commands, that is to say, it is executed as if it were typed from the console.

An attempt to execute either a file without execute permission or a directory results in a diagnostic message, as does the inablility to find the file.

Pipe-lines. Commands separated by '|' or '^' constitute a sequence of processes called a pipeline. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected to its neighbors by pipes (see pipe(II)).

Two or more commands may appear on the same line in other ways. When they are separated by ';', the commands are executed in order of appearance on the line. However, if a command is followed by '&', *sh* does not wait for the completion of that process, but reports the process-id of the executing process, and continues with the next command.

Commands on a line may be grouped by using parentheses. To illustrate:

```
(pwd;date) | lpr
```

sends the name of the current directory and the date to the line printer.

Redirection of I/O. Three files are opened by *sh.* A standard input file (file descriptor 0) from which input lines are read; a standard output file (file descriptor 1) on which printed output can be written; and a standard error output file (file descriptor 2) on which diagnostics can be written. These files normally point to the user's terminal but can be redirected to other devices or files.

There are five redirection symbols that cause the immediately following string to be interpreted as a special argument to the Shell itself. Such an argument may appear anywhere among the arguments of a command, or before or after a parenthesized command list, and is associated with that command or command list.

<name	causes the file 'name' to be used as the standard input of the associated command.
>name	causes file 'name' to be used as the standard output for the associated command. 'Name' is created if it did not exist, and is truncated at the outset. When '%' is used as 'name', the output is directed to the standard error output file.
>>name	causes file 'name' to be used as the standard output for the associated command. If 'name' did not exist, it is created; if it did exist, the command output is appended to the file.

As an example, either of the command lines

```
ls >junk; cat tail >>junk
( ls; cat tail ) >junk
```

creates, on file 'junk', a listing of the working directory, followed immediately by the contents of file 'tail'.

Either of the constructs '>name' or '>>name' associated with any but the last command of a pipe-line is ineffectual, as is '<name' in any but the first.

%name	causes 'name' to be used as the error output file for the associated command. 'Name' is created if it did not exist and is truncated at the outset.
%%name	causes 'name' to be used as the error output for the associated command. If 'name' did not exist, it is created; if it existed, the error output is appended to the file.

Generation of argument lists. If any argument contains any of the characters '?', '*' or '[', it is treated specially as follows. The directory is searched for files which *match* the given argument according to the following rules:

1. The character '*' in an argument matches any string of characters in a file 'name' (including the null string).
2. The character '?' matches any single character in a file name.
3. Square brackets '[...]' specify a class of characters which matches any single file-name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by '–' places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.
4. Other characters match only the same character in the file name.

For example, '*' matches all file names; '?' matches all one-character file names; '[ab]*.s' matches all file names beginning with 'a' or 'b' and ending with '.s'; '?[zi–m]' matches all two-character file names ending with 'z' or the letters 'i' through 'm'.

If the argument with '*' or '?' also contains a '/', a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the argument up to the last '/' before a '*' or '?'. The matching process matches the remainder of the argument after this '/' against the files in the derived directory. For example: '/usr/dmr/a*.s' matches all files in directory '/usr/dmr' which begin with 'a' and end with '.s'.

By this procedure a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the '*', '[', or '?'. The same process is carried out for each argument (the resulting lists are *not* merged) and finally the command is called with the resulting list of arguments.

Quoting. The character '\' causes the immediately following character to lose any special meaning it may have to the Shell; in this way '<', '>', and other characters meaningful to the Shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by '\' is translated into a blank.

Quoting may be accomplished by double (") or single (′) quotes in the following way:

```
ls | pr −h "My directory"
```

causes a directory listing to be produced by *ls,* and passed on to *pr* to be printed with the heading 'My directory'. Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr.* Characters enclosed in double quotes have no special significance to *sh,* which passes them along as an argument with no interpretation. Within single quotes, *sh* expands '\' and '$', and takes all other characters literally.

Command Files. Since *sh* treats its first argument as a command, that argument may itself be *sh;* thus giving the Shell recursive power. A file of commands to the Shell may be prepared prior to execution of *sh.* Either of the commands

```
sh < file
sh file
```

will cause the execution of the file of commands.

Sh becomes a simple but powerful programming language by the inclusion of the following commands that have meaning within a command file:

goto	supplies the branch facility. (see *goto* (I))
exit	terminates a shell. (see *exit* (I))
: label	locates the destination of a branch. The line itself is meaningless to *sh* and is ignored.
return	transfers control within a command file. (see *return* (I))
shift	manipulates the arguments of a command file. (see *shift* (I))
if	a conditional statement. (see *if* (I))
onintr	identifies the routine to be used for interrupt processing. (see *onintr* (I))
read	along with onend, and open, provides a facility for accessing information sequentially from a file outside the command file. (see *read* (I))
echo	writes data to the standard output. (see *echo* (I))
= x [value]	makes an assignment of *value* to a single letter variable within a command file. If *value* is not specified, the variable takes on the value of the entire next line of the standard input. These variables once assigned, may be referenced throughout the command file as '$a','$b',...'$z'

Certain other variables may be referenced within a command file:

$0	the name of the command file. When no name exists, $0 is null.
$$	the process-id of the executing shell, expressed in five decimal digits (with leading zeroes).
$.	the pathname of the directory from which the command file was executed.
$T	the pathname of the controlling terminal.
$E	an octal representation of the return code of the most recently completed process in the file. It is useful with the *if* statement.
$R	same value as $E in decimal representation.

Argument passing. When *sh* is invoked as a command, it has additional string processing capabilities. Recall that the form in which the Shell is invoked is

```
sh [ name [ arg1 ... [ arg9 ] ] ]
```

Name is a file which is read and interpreted. If not given, this subinstance of the Shell continues to read the standard input file.

In command lines in the file (not in command input), character sequences of the form '$n', where *n* is a digit, are replaced by the *n*th argument to the invocation of the Shell (arg*n*). '$0' is replaced by *name.* Up to nine arguments can be referenced at any one time.

Optional arguments which may be supplied to *sh.*

−t	Used alone, it causes *sh* to read a single line from the standard input, execute it as a command, and then exit. It is useful for interactive programs which allow users to execute system commands. For example, see'!' in *bc* (I) and *ed* (I).
−c	Used with one following argument it causes the next argument to be taken as a command line and executed. No new-line need be present, but new-line characters are treated appropriately. It is useful as an alternative to −t where the caller has already read some of the characters of the command to be executed.
−	A new shell process is invoked which reads from the standard input. No prompts are issued in this environment.
−v	When the verbose option is selected, *sh* will echo all command lines as they are executed. The feature is useful in debugging command files.
−i	Interrupts are ignored by *sh.* Ordinarily interrupts terminate a command file process.

If a user supplies a command file named *profile.sh* in his login directory, *sh* executes that command file at its first invocation. This is useful for setting up a specific environment in which to work.

End of file. An end-of-file in the Shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an EOT.

Special commands. The following commands are treated specially by the Shell.

chdir	is done without spawning a new process by executing *sys chdir* (II).
login	is done by executing /bin/login without creating a new process.

wait	is done without spawning a new process by executing *sys wait* (II).
exit	is done without spawning a new process by executing *sys exit* (II).
return	is done without spawning a new process, either by calling *sys exit* or by returning to standard Shell input.
shift	is done by manipulating the arguments to the command file.
=	is done by storing the assigned value in a dynamic buffer that increases in size with each new assignment.
read *open* *onend*	are all done without spawning a new process so the Shell can keep its place in the file being read.
onintr	is done without spawning a new process for the interrupt processing routine to handle trap resets.

Command file errors; interrupts. Any Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file.

Processes that are created with '&' ignore interrupts. In addition, if such a process has not redirected its input with a '<', its input is automatically redirected to the zero length file /dev/null.

Termination Reporting. If a command (not followed by '&') terminates abnormally, a message is printed. (All terminations other than exit and interrupt are considered abnormal.) Termination reports for commands followed by '&' are given upon receipt of the first command subsequent to the termination of the command, or when a *wait* is executed. The following is a list of the abnormal termination messages:

Bus error
Trace/BPT trap
Illegal instruction
IOT trap
EMT trap
Bad system call
Quit
Floating exception
Memory violation
Killed
Broken Pipe
Alarm timeout

If a core image is produced, '– Core dumped' is appended to the appropriate message.

FILES

/etc/glob, which interprets '*', '?' and '['.
/dev/null as a source of end-of-file.
/dev/ln??, the control terminal.
$./profile.sh, to set an individual user's environment.

SEE ALSO

'The UNIX Time-Sharing System', CACM, July, 1974, which gives the theory of operation of the Shell.
chdir (I), login (I), wait (I), shift (I), goto (I), read (I), return (I), exit (I), onintr (I), glob (VIII), echo (I), pipe (II).

DIAGNOSTICS

'Too many tokens','Command table overflow', and 'Too many characters', indicate buffer overflow. The command line is terminated.
''$' nesting error'; if more than nine levels of arguments occurs. The command line is terminated.

BUGS

Too many variable assignments in a command file may result in buffer overflow. Deleting out of a command may cause the shell to die if you have a *profile.sh* file.

SHIFT (I) SHIFT (I)

NAME

shift − adjust Shell arguments

SYNOPSIS

shift [n]

DESCRIPTION

Shift is used in command files to shift the argument list left by 1, so that old *$2* can now be referred to by *$1* and so forth. *Shift* is useful to iterate over several arguments to a command file. For example, the command file

```
: loop
if $1x = x exit
pr −3 $1
shift
goto loop
```

prints each of its arguments in 3-column format.

The optional argument, **n** leaves the first n-1 arguments untouched, and replaces arg n with arg n+1, arg n+1 with arg n+2, ... through the remainder of the argument list.

Shift is executed within the Shell.

SEE ALSO

sh (I)

BUGS

SIZE (I) SIZE (I)

NAME

size − size of an object file

SYNOPSIS

size [object ...]

DESCRIPTION

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in octal and decimal, of each object-file argument. If no file is specified, **a.out** is used.

BUGS

SLEEP (I)

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command in a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often as in this shell command file:

```
: loop
command
sleep 37
goto loop
```

SEE ALSO

sleep (II)

BUGS

Time must be less than 65536 seconds.

SORT (I) SORT (I)

NAME

sort − sort or merge files

SYNOPSIS

sort [**−mubdfinrt***x*] [**+***pos* [**−***pos*]] ... [**−o** name] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '−' means the standard input. The standard input is also used if no input file names are given. Thus *sort* may be used as a filter.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected by the following flags one or more of which may appear.

b	Leading blanks (spaces and tabs) are not included in keys.
d	'Dictionary' order: only letters, digits and blanks are significant in comparisons.
f	Fold lower case letters onto upper case.
i	Ignore all nonprinting nonblank characters in nonnumeric comparisons.
n	An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.
r	Reverse the sense of comparisons.
t*x*	Tab character starting fields is *x*.

Selected parts of the line, specified by **+***pos* and **−***pos*, may be used as sort keys. *Pos* has the form *m.n* optionally followed by one or more of the flags **bdfinr,** where *m* specifies a number of fields to skip, *n* a number of characters to skip further into the next field, and the flags specify a special ordering rule for the key. A missing *.n* is taken to be 0. **+***pos* denotes the beginning of the key; **−***pos* denotes the first position after the key (end of line by default). Later keys are compared only when all earlier keys compare equal.

When no tab character has been specified, a field consists of nonblanks and any preceding blanks. Under the **−b** flag, leading blanks are excluded from a field. When a tab character has been specified, fields are strings separated by tab characters.

Lines that otherwise compare equal are ordered with all bytes significant.

These flag arguments are also understood:

−m	Merge only, the input files are already sorted.
−o	The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs, except under the merge flag **−m**.
−u	Suppress all but one in each set of contiguous equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print a list of all the distinct *nroff* (I) commands in a given document:

```
grep "^\." document | sort −u +0 −0.3
```

Print the password file *passwd* (V) sorted by user id:

```
sort −t: +2n /etc/passwd
```

FILES

/usr/tmp/stm???

BUGS

SPLIT (I) SPLIT (I)

NAME

split − split a file into pieces

SYNOPSIS

split [−*n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if **−** is given in its stead, then the standard input file is used.

BUGS

STRIP (I) STRIP (I)

NAME

strip — remove symbols and relocation bits

SYNOPSIS

strip name ...

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the the same as use of the **−s** option of *ld.*

FILES

/tmp/stm?	temporary file

SEE ALSO

ld (I), as (I)

BUGS

NAME

stty − set typewriter options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output typewriter. With no argument, it reports the current settings of the options. The option strings are selected from the following set:

even	allow even parity
−even	disallow even parity
odd	allow odd parity
−odd	disallow odd parity
raw	raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
−raw	negate raw mode
−nl	allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl	accept only new-line to end lines
echo	echo back every character typed
−echo	do not echo characters
lcase	map upper case to lower case
−lcase	do not map case
−tabs	replace tabs by spaces when printing
tabs	preserve tabs
ek	use '#' and '@' as erase and kill characters, respectively
erase char	use 'char' as erase character
kill char	use 'char' as kill character
cr0 cr1 cr2 cr3	select style of delay for carriage return (see *stty* (II))
nl0 nl1 nl2 nl3	select style of delay for linefeed (see *stty* (II))
tab0 tab1 tab2 tab3	select style of delay for tab (see *stty* (II))
ff0 ff1	select style of delay for form feed (see *stty* (II))
tty33	set all modes suitable for Teletype model 33
tty37	set all modes suitable for Teletype model 37
vt05	set all modes suitable for DEC VT05 terminal
tn300	set all modes suitable for GE Terminet 300
ti700	set all modes suitable for Texas Instruments 700 terminal
tek	set all modes suitable for Tektronix 4014 terminal
50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb	Set typewriter baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

The various delay algorithms are tuned to various kinds of terminals. In general the specifications ending in '0' mean no delay for the corresponding character.

SEE ALSO

stty (II)

SUM (I) SUM (I)

NAME

sum — sum file

SYNOPSIS

sum name ...

DESCRIPTION

Sum sums the contents of the bytes (mod 2^16) of one or more files and prints the answer in decimal. A separate sum is printed for each file specified, along with the number of whole or partial 512-byte blocks read.

In practice, *sum* is often used to verify that all of a special file can be read without error.

BUGS

TAIL(I) TAIL(I)

NAME

tail − deliver the last part of a file

SYNOPSIS

tail [±number[**lbc**]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or −*number* from the end of the input. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c.** When no units are specified, counting is by lines.

SEE ALSO

dd(I)

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tee – pipe fitting

SYNOPSIS

tee [name ...]

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the named files.

BUGS

TIME (I)

NAME

time – time a command

SYNOPSIS

time command

DESCRIPTION

The given command is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command.

The execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

BUGS

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

tp — manipulate DECtape and magtape

SYNOPSIS

tp [key] [name ...]

DESCRIPTION

Tp saves and restores files on DECtape or magtape. Its actions are controlled by the *key* argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped, restored, or listed. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r	The named files are written on the tape. If files with the same names already exist, they are replaced. 'Same' is determined by string comparison, so './abc' can never be the same as '/usr/dmr/abc' even if '/usr/dmr' is the current directory. If no file argument is given, '.' is the default.
u	Update the tape. **u** is like **r,** but a file is replaced only if its modification date is later than the date stored on the tape; that is to say, if it has changed since it was dumped. **u** is the default command if none is given.
d	Delete the named files from the tape. At least one name argument must be given. This function is not permitted on magtapes.
x	Extract the named files from the tape to the file system. The owner and mode are restored. If no file argument is given, the entire contents of the tape are extracted.
t	List the names of the specified files. If no file argument is given, the entire contents of the tape is listed.

The following characters may be used in addition to the letter which selects the function desired.

m	Specifies magtape as opposed to DECtape.
0,...,7	This modifier selects the drive on which the tape is mounted. For DECtape, 'x' is default; for magtape '0' is the default.
v	Normally *tp* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.
c	A fresh dump is to be created; the tape directory is zeroed before beginning. Usable only with **r** and **u.** This option is assumed with magtape since it is impossible to selectively overwrite magtape.
f	causes new entries on tape to be 'fake' in that no data is present for these entries. Such fake entries cannot be extracted. Usable only with **r** and **u.**
i	Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.
w	causes *tp* to pause before treating each file, type the indicative letter and the file name (as with v) and await the user's response. Response **y** means 'yes', so the file is treated. Null response means 'no', and the file does not take part in whatever is being done. Response **x** means 'exit'; the *tp* command terminates immediately. In the **x** function, files previously asked about have been extracted already. With **r, u,** and **d** no change has been made to the tape.

FILES

/dev/tap?
/dev/mt?

DIAGNOSTICS

Several; the non-obvious one is ‘Phase error’, which means the file changed after it was selected for dumping but before it was dumped.

BUGS

A single file with several links to it is treated like several files.

TR(I) TR(I)

NAME

tr − transliterate

SYNOPSIS

tr [**−cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*.

Any combination of the options **−cds** may be used.

- **−c** complements the set of characters in *string1* with respect to the universe of characters whose ascii codes are 001 through 377 octal.
- **−d** deletes all input characters in *string1*.
- **−s** squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[*a*−*b***]** stands for the string of characters whose ascii codes run from character *a* to character *b.*

[*a****n***]**, where *n* is an integer or empty, stands for *n*-fold repetition of character *a*. *n* is taken to be octal or decimal according as its first digit is or is not zero. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character '\' may be used as in *sh* to remove special meaning from any character in a string. In addition, '\' followed by 1, 2 or 3 octal digits stands for the character whose ascii code is given by those digits.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the Shell; 012 is the ascii code for newline.

```
tr −cs "[A−Z][a−z]" "[\012*]" <file1 >file2
```

SEE ALSO

sh (I), ed (I), ascii (VII)

BUGS

Won't handle ascii NUL in *string1* or *string2;* always deletes NUL from input.

NAME

troff – Format text for the Graphic Systems Phototypesetter

SYNOPSIS

troff [*options*] *files*

DESCRIPTION

TROFF accepts lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program.

An argument consisting of a single minus (**–**) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
–o*list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N–M* and means pages *N* through *M;* an initial *–N* means from the beginning to page *N;* and a final *N–* means from *N* to the end.
–n*N*	Number first generated page *N*.
–s*N*	Stop every *N* pages. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
–m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files.*
–r*aN*	Register *a* (one-character) is set to *N.*
–i	Read standard input after the input files are exhausted.
–q	Invoke the simultaneous input-output mode of the **rd** request.
–t	Direct output to the standard output instead of the phototypesetter.
–f	Refrain from feeding out paper and stopping phototypesetter at the end of the run.
–w	Wait until phototypesetter is available, if currently busy.
–b	TROFF will report whether the phototypesetter is busy or available. No text processing is done.
–a	Send a printable (ASCII) approximation of the results to the standard output.
–p*N*	Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.
–g	Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta00000	temporary file
/usr/lib/tmac.*	standard macro files
/usr/lib/font/*	font width tables

SEE ALSO

NROFF/TROFF User's Manual (internal memorandum).
Troff Made Trivial (internal memorandum).
tbl(I), eqn(I)

BUGS

NAME

typo — find possible typos

SYNOPSIS

typo [**−1**] [**−n**] file ...

DESCRIPTION

Typo hunts through a document for unusual words, typographic errors, and *hapax legomena* and prints them on the standard output.

The words used in the document are printed out in decreasing order of peculiarity along with an index of peculiarity. An index of 10 or more is considered peculiar. Printing of certain very common English words is suppressed.

The statistics for judging words are taken from the document itself, with some help from known statistics of English. The **−n** option suppresses the help from English and should be used if the document is written in, for example, Urdu.

The **−1** option causes the final output to appear in a single column instead of three columns. The normal header and pagination is also suppressed.

Nroff (I) control lines are ignored. Upper case is mapped into lower case. Quote marks, vertical bars, hyphens, and ampersands within words are equivalent to spaces. Words hyphenated across lines are put back together.

FILES

/tmp/ttmp??
/usr/lib/salt
/usr/lib/w2006

BUGS

Because of the mapping into lower case and the stripping of special characters, words may be hard to locate in the original text.

UNIQ (I) UNIQ (I)

NAME

uniq − report repeated lines in a file

SYNOPSIS

uniq [**−udc** [**+**n] [**−**n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see sort(I). If the **−u** flag is used, just the lines that are not repeated in the original file are output. The **−d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **−u** and **−d** mode outputs.

The **−c** option supersedes **−u** and **−d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

−*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

sort (I), comm (I)

BUGS

WAIT (I) WAIT (I)

NAME

wait − await completion of process

SYNOPSIS

wait [cpid]

DESCRIPTION

Wait waits until all processes started asynchronously (with '&') have terminated. If a child process-id (cpid) is specified as an argument, then the wait is only for termination of that asynchronous process.

SEE ALSO

sh (I)

BUGS

If *cpid* is not an asynchronous process, then *wait* waits for the termination of all asynchronous processes. It should indicate that *cpid* is invalid.

NAME

wall – write to all users

SYNOPSIS

wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by "Broadcast Message from *yourname* (ln??)...". It is used to warn all users, typically prior to shutting down the system.

The sender should be super-user to override any protections the users may have invoked.

FILES

/dev/ln??

SEE ALSO

mesg (I), write (I)

DIAGNOSTICS

"Cannot send to ..." when the open on a user's terminal file fails.

BUGS

NAME

wc − word count

SYNOPSIS

wc [−] [name ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of printing characters delimited by spaces, tabs or newlines. All other characters are simply ignored.

The − argument gives only the word count of the file.

BUGS

WHAT(I) WHAT(I)

NAME

what – identify SCCS files

SYNOPSIS

what name ...

DESCRIPTION

What searches the given files for all occurrences of the pattern which *get* (I) substitutes for %Z% (this is ˜|^` or @(#) at this printing) and prints what follows until the first double quote, greater than, newline, or null character. For example, if the C program in file `f.c′ contains

```
char iden__[] " ˜|^` identification information";
```

or

```
char iden__[] "@(#) identification information";
```

and f.c is compiled to yield ’f.o’ and ’a.out’ then the command

```
what f.c f.o a.out
```

will print

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

What is intended to be used in conjunction with the SCCS command *get* (I), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(I), help(I)

DIAGNOSTICS

Use *help* (I) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern ˜|^` or @(#) could be found just by chance, but this causes no harm in nearly all cases.

NAME

who – who is on the system

SYNOPSIS

who [**-ulpdbra**] [file]

who am i

DESCRIPTION

Who, without an argument announces who is on the system. It lists the user-id, login time, and the process-id.

Who am i identifies the user.

Without an argument, *who* examines the /etc/utmp file. *Who* **file**, alternately processes the data of the file. Typically the given file will be /etc/wtmp, which contains a record of all logins since it was created.

Flags are processed by *who* and produce the following listings:

u	Logged on user.
l	Connected lines not logged on.
p	Currently active processes not associated with a line.
d	All expired processes not respawned by the *init*(VIII) process.
b	Last instance of hardware boot.
r	Current *run state* of *init*(VIII).
a	Print all options.

The name LOGIN in the *who* outputs means a login is in progress; ****** indicates a defunct process; BOOT indicates a UNIX reboot; RUN STATE reflects the *init*(VIII) run state. Execution of *date*(I) to change time results in a record of old system time, OT, and new system time, NT. Run state changes are indicated with RC.

FILES

/etc/utmp	record of users on system
/etc/wtmp	logins, logouts, reboots, modification of system time and action by *init*(VIII)
/etc/lines	defines action to be taken by *init*(VIII)

SEE ALSO

login(I), date(I), init(VII)

BUGS

WRITE(I) WRITE(I)

NAME

write – write to another user

SYNOPSIS

write user [lineno]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message

message from *yourname* (ln??)...

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point *write* writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *lineno* argument may be used to indicate the last character(s) of the appropriate terminal name; otherwise the first instance of the user found in the who (*/etc/utmp*) file is assumed and the following message posted:

user multiply logged in
writing to ln??

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *nroff* and *pr,* disallow messages in order to prevent messy output. However, if the user has super user permissions messages can be forced onto a write inhibited terminal.

If the character '!' is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write:* when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal (**(o)** for 'over' is conventional) that the other may reply. **(oo)** (for 'over and out') is suggested when conversation is about to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute '!'

SEE ALSO

mesg (I), who (I), mail (I)

DIAGNOSTICS

user not logged in

BUGS

YACC (I) YACC (I)

NAME

yacc − yet another compiler-compiler

SYNOPSIS

yacc [**−vo**] [grammar]

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output is *y.tab.c,* which must be compiled by the C compiler and loaded with any other routines required (perhaps a lexical analyzer) and the Yacc library:

```
cc y.tab.c other.o −ly
```

If the **−v** flag is given, the file *y.output* is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

The **−o** flag calls an optimizer for the tables; the optimized tables, with parser included, appear on file *y.tab.c*

SEE ALSO

''LR Parsing'', by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974. ''The YACC Compiler-compiler'', UNIX handbook.

AUTHOR

S. C. Johnson

FILES

y.output	
y.tab.c	
yacc.tmp	when optimizer is called
/lib/liby.a	runtime library for compiler
/usr/source/yacc/opar.c	parser for optimized tables
/usr/yacc/yopti	optimizer postpass

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the *y.output* file.

BUGS

Because file names are fixed, at most one Yacc process can be active in a given directory at a time.

INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions *bes* and *bec* (‘‘branch on error set (or clear)’’). These are synonyms for the *bcs* and *bcc* instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is −1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable *errno* is set to the error number. *Errno* is not cleared on successful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III)*.

The possible error numbers are not recited with each writeup in section II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror*. A short explanation is also provided.

0 - (unused)

1 EPERM Not owner and not super-user
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn’t, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to *signal* does not exist, or is already dead. Also, attempting to send a message to a process that has not enabled message reception.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or is beyond the limits of the allowed number of subdevices. It may also occur when, for example, a tape drive is not dialed in

or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes (counting the null at the end of each argument) is presented to *exec.*

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407 or 410.

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

10 ECHILD No children
Wait was requested but the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a *fork,* the system's process table is full and no more processes can for the moment be created.

12 ENOMEM Not enough core
During an *exec* or *break,* a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments is such as to require more than the existing 8 segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file or some other resource in a way forbidden by the protection system.

14 EFAULT Memory fault
User has supplied a non-existent address.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in *mount.*

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an open file or some process's current directory.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. *link.*

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir.*

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: currently, dismounting a non-mounted device, mentioning an unknown signal in *signal,* giving an unknown request in *stty* to the TIU special file, passing an invalid argument list to *exec,* and providing an unknown *function* argument to *lock* or *msg.*

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Only 15 files can be open per process.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a typewriter or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
An attempt to make a file larger than the maximum of 32768 blocks.

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Seek on pipe
A *seek* was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 127 links to a file.

32 EPIPE Write on broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 ETABLE No entries left
One of the system tables neccessary to complete the request is temporarily full.

34 EFUNC Invalid function
An attempt to perform an invalid operation (e.g. trying to *unlock* a semaphore not locked by the

invoking process).

35 ENOMSG No message
A message of the requested type is not on the message queue.

36 ENOALOC Resource not allocated
An attempt was made either to use a facility that must first be allocated (e.g. *recvw* without first enabling messages) or to allocate a facility in a way other than for its intended use (e.g. trying to allocate a P-V semaphore for lock-unlock use).

37 EBIOCTL Invalid ioctl operation
ioctl operation has been rejected by device driver. This generally results from an improper device specification or command.

ACCESS(II) ACCESS(II)

NAME

access − determine accessibility of file

SYNOPSIS

(access = 33.; not in assembler)
sys access; name; mode

access(name, mode)
char *name;

DESCRIPTION

Access checks the given file *name* for accessibility according to *mode,* which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. An appropriate error indication is returned if one or more of the desired access modes is not permitted.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by *access,* but an attempt to open it for writing will fail (although files may be created there); a file may look excutable, but *exev* will fail unless it is in proper format.

DIAGNOSTICS

C-bit is set on disallowed accesses, and the error code appears in r0. From C, −1 is returned and the error code appears in *errno.* 0 is returned from successful tests.

BUGS

ACCT(II) ACCT(II)

NAME

acct — turn accounting on/off

SYNOPSIS

(acct = 51.; not in assembler)
sys acct; name

acct(name)
char *name

DESCRIPTION

System per process accounting is started and stopped by the *acct* system call. When system accounting is active, a record in shell accounting format (*sa*(VIII)) is placed at the end of the file *name* for each process that terminates. *Name* is a pointer to a null terminated string of ASCII characters that represents the file's pathname. System accounting is terminated by invoking *acct* with a zero value for *name.*

Only the super-user may start and stop system accounting.

SEE ALSO

accton (VIII), sa (VIII)

DIAGNOSTICS

The error bit (c-bit) is set if the user is not super-user or if an attempt is made to start accounting when it is already active. From C, a −1 return indicates an error.

NAME

alarm − activate alarm clock timer

SYNOPSIS

(alarm = 27.; not in assembler)
(seconds in r0)
sys alarm

alarm(seconds)

DESCRIPTION

Alarm returns immediately but starts a timer within the system which causes the alarm clock signal to be sent to the current process after the specified number of seconds have elapsed. The changing of the date via stime(II) does not cause the signal to be posted prematurely.

In assembly language, the old value of the timer (in seconds) is returned in r0. In C, that value is returned.

SEE ALSO

pause(II), sleep(II)

DIAGNOSTICS

—

BREAK (II) BREAK (II)

NAME

break, brk, sbrk – change core allocation

SYNOPSIS

(break = 17.)
sys break; addr

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Break sets the system's idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes). Locations greater than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

From C, *brk* will set the break to *addr.* The old break is returned.

In the alternate entry *sbrk, incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break.*

SEE ALSO

exec (II), alloc (III), end (III)

DIAGNOSTICS

The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break. From C, −1 is returned for these errors.

BUGS

Setting the break in the range 0177700 to 0177777 is the same as setting it to zero.

CALL(II) CALL(II)

NAME

call, lcall, vcall − create and execute a new process

SYNOPSIS

```
(call = 57.)
sys call; name; args
...
name: <...\0>
...
 args: arg0; arg1; ...; 0
 arg0: <...\0>
 arg1: <...\0>
   ...

lcall(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

vcall(name, argv)
char *name;
char *argv[ ];
```

DESCRIPTION

Call performs a combined *fork* and *exec* in one step. The newly created child process begins execution at the beginning of the core image of the file and the parent process continues execution without waiting for the child to terminate. *Call* returns the process id of the newly created process.

Files remain open across *call* calls. All signals are reset to the system's default value when *call* is executed.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. *Call* changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The real user ID is not affected.

The first argument to *call* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as in the *exec* system call.

From C, two interfaces are available, *lcall* and *vcall. lcall* is useful when a known file with known arguments is being called; the arguments to *lcall* are the character strings constituting the file name and the arguments. As in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list. The *vcall* version is useful when the number of arguments is unknown in advance; the arguments to *vcall* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

SEE ALSO

fork (II), exec(II)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 10 * 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1 on error. The process id of the new process is returned on a success.

NAME

chdir − change working directory

SYNOPSIS

(chdir = 12.)
sys chdir; dirname

chdir(dirname)
char *dirname;

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes this directory to become the current working directory.

SEE ALSO

chdir (I)

DIAGNOSTICS

The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C, a −1 returned value indicates an error, 0 indicates success.

NAME

chmod − change mode of file

SYNOPSIS

(chmod = 15.)
sys chmod; name; mode

chmod(name, mode)
char *name;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode.* Modes are constructed by ORing together some combination of the following:

4000 set user ID on execution
2000 set group ID on execution
1000 save text image after execution
0400 read by owner
0200 write by owner
0100 execute (search on directory) by owner
0070 read, write, execute (search) by group
0007 read, write, execute (search) by others

Only the owner of a file (or the super-user) may change the mode.

SEE ALSO

chmod (I)

DIAGNOSTIC

Error bit (c-bit) set if *name* cannot be found or if current user is neither the owner of the file nor the super-user. From C, a −1 returned value indicates an error, 0 indicates success.

CHOWN (II)

NAME

chown – change owner and group of a file

SYNOPSIS

(chmod = 16.)
sys chown; name; owner

chown(name, owner)
char *name;

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its owner and group changed to the low and high bytes of *owner* respectively. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

SEE ALSO

chown (VIII), passwd (V)

DIAGNOSTICS

The error bit (c-bit) is set on illegal owner changes. From C a −1 returned value indicates error, 0 indicates success.

CHROOT(II) CHROOT(II)

NAME

chroot − change root directory

SYNOPSIS

(chroot = 60.; not in assembler)
sys chroot; dirname

chroot(rootname)
char *rootname;

DESCRIPTION

Rootname is the address of the pathname of a directory, terminated by a null byte. *Chroot* causes this directory to become the process's root directory. This means that any references to file names beginning with slash are not relative to the real root of the UNIX file system, but relative to the new root directory specified in this system call. The current working directory remains unchanged. A *chdir*(I) to a slash ("/") following the *chroot* system call will change the working directory to the new root directory. Arguments to *chroot* are always absolute: no special meaning is given to initial slashes even if a *chroot* is currently in effect.

This system call is restricted to the super user.

SEE ALSO

chroot(I)

DIAGNOSTICS

The error bit (c-bit) is set if the given name is not that of a directory or is not searchable (executable) or the current user is not the super user. From C, a −1 returned value indicates an error, 0 indicates success.

CLOSE (II) CLOSE (II)

NAME

close – close a file

SYNOPSIS

(close = 6.)
(file descriptor in r0)
sys close

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open, creat,* or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit,* but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

SEE ALSO

creat (II), open (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set for an unknown file descriptor. From C a −1 indicates an error, 0 indicates success.

NAME

creat — create a new file

SYNOPSIS

(creat = 8.)
sys creat; name; mode
(file descriptor in r0)

creat(name, mode)
char *name;

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name,* given as the address of a null-terminated string. If the file did not exist, it is given mode *mode.* See *chmod* (II) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned (in r0).

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat,* an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write (II), close (II), stat (II)

DIAGNOSTICS

The error bit (c-bit) may be set if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

From C, a −1 return indicates an error.

NAME

csw − read console switches

SYNOPSIS

(csw = 38.; not in assembler)
sys csw

getcsw( )

DESCRIPTION

The setting of the console switches is returned (in r0).

DUP(II) DUP(II)

NAME

dup − duplicate an open file descriptor

SYNOPSIS

(dup = 41.; not in assembler)
(file descriptor in r0)
sys dup

dup(fildes)
int fildes;

DESCRIPTION

Given a file descriptor returned from an *open, pipe,* or *creat* call, *dup* will allocate another file descriptor synonymous with the original. The new file descriptor is returned in r0. Normally the first available file descriptor is returned, however, the system can be forced to assign file descriptors starting at some number other than zero by setting the high byte of *fildes* to the desired starting point. The system will then attempt to use a file descriptor equal to this value of larger. If all of the file descriptors beyond the start point are used, the user is returned an error even if there are file descriptors with smaller values available. A process may have up to 20 file descriptors open at a time and the file descriptors will be assigned as numbers from zero to nineteen. The chief advantage of forcing the assignment of a file descriptor to begin at a particular value is that under the current implementation of *exec* and *call* the last five file descriptors are automatically closed and therefore cannot be inherited by the overlaid process. Forcing a file descriptor to be allocated as one of the last five eliminates the need to close that file beforedoing an overlay. (This is particularly advantageous for the *shell.)*

Dup is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor. Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way. This is handy for manipulating standard input and/or standard output files.

SEE ALSO

creat (II), open (II), close (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set if the given file descriptor is invalid, there are already too many open files, or if there are no more file descriptors beyond the value specified in the high byte of *fildes.* From C, a −1 returned value indicates an error.

NAME

errlog – error logging

SYNOPSIS

(errlog = 50.; not in assembler)
sys errlog; flag

errlog(flag)

DESCRIPTION

Error logging is started and stopped by the *errlog* system call. A value of one for the *flag* argument activates logging, while a value of zero terminates it. Zero is returned upon successful completion of these calls. Errors that are to be logged are transmitted to the logging process by means of messages, so the logger must have message reception enabled before turning on logging. A startup and, if possible, a stop message are sent to the logger immediately after logging is activated and terminated, respectively. Logging is stopped in one of three ways: by calling *errlog* with *flag* zero, disabling message reception, or by doing an *exit.*

Use of *errlog* is restricted to the super-user. Only one error logging process may be active at any time.

SEE ALSO

msg (II), errdemon (VIII)

DIAGNOSTICS

The error bit (c-bit) is set on error. Typically, this occurs if the user is not super-user, if error logging is active for another process, if message reception is not enabled, or if a process attempts to terminate logging that was started on behalf of another process. From C, a –1 return indicates an error.

NAME

exec, execl, execv − execute a file

SYNOPSIS

```
(exec = 11.)
sys exec; name; args
...
name: <...\0>
...
args: arg0; arg1; ...; 0
arg0: <...\0>
arg1: <...\0>
   ...

execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name;
char *argv[ ];
```

DESCRIPTION

Exec overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec,* but signals that are caught are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. *Exec* changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The real user ID is not affected.

The form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as follows. The stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings. The arguments are placed as high as possible in core.

```
 sp→   nargs
       arg0
       ...
       argn
       −1

arg0:  <arg0\0>
       ...
argn:  <argn\0>
```

From C, two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments; as in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv)
int argc;
char **argv;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is not directly usable in another *execv,* since *argv[argc]* is −1 and not 0.

SEE ALSO

fork (II)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 5120 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

NAME

exit − terminate process

SYNOPSIS

```
(exit = 1.)
(status in r0)
sys exit

exit(status)
int status;
```

DESCRIPTION

Exit is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait.* The low byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

SEE ALSO

wait (II)

DIAGNOSTICS

None.

NAME

fork − spawn new process

SYNOPSIS

(fork = 2.)
sys fork
(new process return)
(old process return)

fork( )

DESCRIPTION

Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of *fork*. The only distinction is the return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. This process ID is used by *wait.*

The two returning processes share all open files that existed before the call. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

From C, the child process receives a 0 return, and the parent receives a non-zero number which is the process ID of the child; a return of −1 indicates inability to create a new process.

SEE ALSO

wait (II), exec (II)

DIAGNOSTICS

The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space. From C, a return of −1 (not just negative) indicates an error.

FSTAT (II) FSTAT (II)

NAME

fstat – get status of open file

SYNOPSIS

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

fstat (fildes, buf)
struct inode *buf;

DESCRIPTION

This call is identical to *stat,* except that it operates on open files instead of files given by name. It is most often used to get the status of the standard input and output files, whose names are unknown.

SEE ALSO

stat (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor is unknown; from C, a −1 return indicates an error, 0 indicates success.

NAME

getgid — get group identifications

SYNOPSIS

(getgid = 47.; not in assembler)
sys getgid

getgid( )

DESCRIPTION

Getgid returns in r0 the real group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set group ID" mode, to find out who invoked them.

SEE ALSO

setgid (II)

DIAGNOSTICS

—

GETPID(II) GETPID(II)

NAME

getpid, getppid — get process identification

SYNOPSIS

(getpid = 20.; not in assembler)
sys getpid
(pid in r0)
(parent pid in r1)

getpid( )
getppid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

Getppid returns the process ID of the parent of the current process. Note that a return of 1 from this call indicates that the parent process has terminated and the current process has been inherited by the *init*(VIII) process

NAME

getuid − get user identifications

SYNOPSIS

(getuid = 24.)
sys getuid

getuid( )

DESCRIPTION

Getuid returns in r0, the real user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the ‘‘set user ID’’ mode, to find out who invoked them.

SEE ALSO

setuid (II)

DIAGNOSTICS

–

GTTY (II) GTTY (II)

NAME

gtty — get typewriter status

SYNOPSIS

(gtty = 32.)
(file descriptor in r0)
sys gtty; arg
...
arg: .=.+6

gtty (fildes, arg)
int arg[3];

DESCRIPTION

Gtty stores in the three words addressed by *arg* the status of the typewriter whose file descriptor is given in r0 (resp. given as the first argument). The format is the same as that passed by *stty*.

SEE ALSO

stty (II)

DIAGNOSTICS

Error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a −1 value is returned for an error, 0, for a successful call.

INDIR (II)

NAME

indir − indirect system call

SYNOPSIS

(indir = 0.; not in assembler)
sys indir; syscall

DESCRIPTION

The system call at the location *syscall* is executed. Execution resumes after the *indir* call.

The main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, the executing process will get a fault.

SEE ALSO

—

DIAGNOSTICS

—

IOCTL(II) IOCTL(II)

NAME

ioctl – I/O control and parameter passing

SYNOPSIS

(ioctl = 54.; not in assembler)

(file descriptor in r0)
sys ioctl; arg1; arg2
...
arg1: .byte command; .byte device
arg2: (dependent on device and command)
ioctl(fildes, (command | (device<<8)), arg2)

DESCRIPTION

ioctl is used to control and pass information to and from device drivers. It is intended as a more general version of *stty(II)* and *gtty(II).*

The file descriptor for the device being referenced is passed in r0 (resp. is the first argument to the call). *device* is a single letter identifying the device driver(s) towards which the call is directed and is passed as a check. Only *'t'* corresponding to actions directed at the low speed asynchronous drivers *(kl, dc* and *dh)* has been thus far defined. The meaning of *command* is dependent on the *device.* The last argument is dependent on the *device* and *command* and may be a pointer. Section IV articles on the various devices give commands and usage.

The calls *gtty(0, arg)* and *stty(0, arg)* are equivalent to *ioctl(0, 010|('t'<<8), arg)* and *ioctl(0, 011|('t'<<8), arg),* respectively.

SEE ALSO

stty(I), stty(II), gtty(II), tty(IV)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a device of the proper type or the command is rejected by the device driver. From C, −1 indicates an error.

KILL (II) KILL (II)

NAME

kill – send signal to a process

SYNOPSIS

(kill = 37.; not in assembler)
(process number in r0)
sys kill; sig

kill(pid, sig);

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number in r0. See signal (II) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes which have the same controlling typewriter and user ID.

In no case is it possible for a process to kill itself.

SEE ALSO

signal (II), kill (I)

DIAGNOSTICS

The error bit (c-bit) is set if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist. From C, −1 is returned.

NAME

link − link to a file

SYNOPSIS

```
(link = 9.)
sys link; name1; name2

link(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A link to *name1* is created; the link has the name *name2.* Either name may be an arbitrary path name.

SEE ALSO

ln (I), unlink (II)

DIAGNOSTICS

The error bit (c-bit) is set when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when more than 127 links are made. From C, a −1 return indicates an error, a 0 return indicates success.

NAME

lock – semaphore operations

SYNOPSIS

(lock = 62.; not in assembler)
sys lock; function; number

allocsem(number, type)
freesem(number)

lock(number)
unlock(number)
tlock(number)

p(number)
v(number)
test(number)

DESCRIPTION

Semaphores are useful for the synchronization of processes and controlling access to resources. There are two versions of semaphores available for these purposes: the lock-unlock type and the P-V (counting) type. A semaphore must be allocated by a process before it may be used; the semaphore's access permission and type are established by the first process to allocate it and remain in effect until freed by every process that has it allocated. Allocated semaphores are inherited across *fork* but not across *exec.*

From assembly language, the low byte of the *function* argument specifies the request type.

0	Lock the semaphore. If it is already locked, suspend execution until it is unlocked (Lock Operation).
1	Unlock the semaphore. Awaken any processes waiting for it to be unlocked (Unlock Operation).
2	Lock the semaphore if it is not already locked. Return immediately if it is locked (Conditional Lock Operation).
3	Decrement the semaphore's value by one if it is positive. If the value is zero, suspend execution until it becomes positive, then decrement it (P Operation).
4	Increment the semaphore's value by one and, if necessary, awaken any processes waiting for the value to become positive (V Operation).
5	Decrement the value of the semaphore by one, but never below zero. Do not suspend execution if the value was already zero (Conditional P operation).
6	Allocate a semaphore to the process. The high byte of *function* indicates how the semaphore is to be used and what class of processes are also eligible to also allocate and use the semaphore. A value of 0, 1, or 2 allocates a lock-unlock semaphore with an access class of anyone, same userid only, and same groupid only, respectively. A value of 3, 4, or 5 allocates a P-V semaphore for use by anyone, same userid only, and same groupid only, respectively.
7	Deallocate (free) the indicated semaphore(s).

The *number* argument designates the semaphore number of the semaphore in question. In allocation requests, a negative value for *number* implies that an available semaphore from the system pool is to be chosen and allocated. A negative value for *number* in deallocation requests indicates that all of the semaphores allocated to the process are to be freed.

After successful calls, r0 contains the semaphore's previous value for lock, unlock, conditional lock, P, V, and conditional P operations; the number of the allocated semaphore for allocation requests; and zero for deallocation requests. Note that a *locked* semaphore's value is the processid of the process that *locked* it; it is zero when not locked.

From C, *allocsem* is used to allocate semaphore *number* for the calling process. If *number* is −1, the first available semaphore from the system pool is allocated. The value of *type* has the same meaning as described above for *function 6.* Semaphore number *number* is deallocated by the *freesem* function. If *number* is −1, all allocated semaphores are freed. The number of the allocated semaphore is returned by *allocsem; freesem* returns zero.

The *lock, unlock,* and *tlock* functions perform lock, unlock, and conditional lock operations, respectively, on the lock-unlock semaphore *number.* Similarly, *p, v,* and *test* perform P, V, and conditional P operations on the P-V semaphore *number.*

Note: From C, if a *lock, unlock, tlock, p, v,* or *test* function is used without the semaphore having been previously allocated by the process, the semaphore will first be allocated before the requested operation is performed.

DIAGNOSTICS

The error bit (c-bit) is set for any one of a number of error conditions. Allocation errors occur if there are no semaphores available, a process tries to allocate too many semaphores, access permission is denied, or the intended usage is incorrect (e.g. allocating a P-V semaphore for lock-unlock use). When attempting to use a semaphore an error occurs when the number is out of range, the semaphore was not previously allocated, or an invalid operation is attempted (e.g. doing a P operation on a lock-unlock semaphore). From C, a −1 return indicates an error.

NAME

mknod − make a directory or a special file

SYNOPSIS

```
(mknod = 14.; not in assembler)
sys mknod; name; mode; addr

mknod(name, mode, addr)
char *name;
```

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. The first physical address of the file is initialized from *addr*. Note that in the case of a directory, *addr* should be zero. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir (I), mknod (VIII), file system (V)

DIAGNOSTICS

Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a −1 value indicates an error.

NAME

mount − mount file system

SYNOPSIS

(mount = 21.)
sys mount; special; name; rwflag

mount(special, name, rwflag)
char *special, *name;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names. Only the super-user may mount a file system.

Name must exist already. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is *0* writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted "read-only" or errors will occur when access times are updated, whether or not any explicit write is attempted. When a mount occurs, an open is issued to the pertinent device driver.

SEE ALSO

mount (VIII), umount (II)

DIAGNOSTICS

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist or is a special file; *special* is already mounted; *name* is in use; there are already too many file systems mounted; *user* is not super-user. From C, a −1 return indicates an error.

NAME

msg − send and receive messages

SYNOPSIS

```
(msg = 49.; not in assembler)
(size in r0)
sys msg; function; buf; arg3; type
msgenab()
msgdisab()

send(buf, size, topid, type)
sendw(buf, size, topid, type)
char *buf;

recv(buf, size, msp, type)
recvw(buf, size, msp, type)
char *buf;
struct mstruct*msp;
```

DESCRIPTION

A process that has enabled message reception has a message queue on which are placed, in order of arrival, messages sent to it by other processes. The process actually receives a message's contents by requesting a message from the queue. A process may send a message to any other process that has enabled message reception, as long as the receiver does not have an excessive number of messages pending on its queue.

From assembly language, the *function* argument specifies the request type.

0 Message reception is disabled; messages may no longer be sent to the process. Depending on the *type,* any message(s) still on the queue are either discarded or returned to the sender. No other arguments are used for this kind of request.

1 Enable message reception. No messages may be sent to the process until this is done. No other arguments are used in this kind of request. Message reception remains enabled across *exec,* but not across *fork.*

2 Send a message to another process. If the system's message buffers are temporarily full, return is immediate. (Conditional Send)

3 Send a message to another process. This is as above, except that execution may be suspended until there is sufficient buffer space to send the message. (Unconditional Send)

4 Receive the first message on the queue of the requested *type.* Return immediately if no such message exists. (Conditional Receive)

5 Receive a message as above, except that execution may be suspended until a suitable message is placed on the queue, if one is not already available. (Unconditional Receive)

The *buf* argument is the address of the buffer that, when sending, contains the message to be sent, or, when receiving, is where the message is to be placed. The number of bytes to be sent or received should be in r0. Currently, messages may be from 0 to 212 bytes in length. If, when receiving, the length of the message exceeds the requested number of bytes, the message is truncated. In any event, the number of bytes actually sent or received is returned in r0.

When a message is being sent, *arg3* should contain the processid of the receiving process. When receiving a message, *arg3* should be the address of a two word buffer into which is placed information about the received message as follows:

```
struct mstruct {
                int             frompid;        /* processid of sender */
                int             mtype;          /* message type */
};
```

The *type* argument is used by a sender to assign a type number (1 to 128) to a message. By convention, types 1 to 63 imply that an acknowledgement message is desired; types 64 to 128 imply no acknowledgement is necessary; type 128 is an acknowledgement message. If a process disables messages (or exits) with any messages still on its queue, those of type 1 to 63 are changed to type 128 and, if possible, returned to the sender; those of type 64 to 128 are discarded.

When receiving messages, a process may request *type* 0, indicating that the first message on the queue is to be retrieved, or a *type* from 1 to 128, indicating that the first message on the queue of the requested *type* is to be received. In either case, the message's actual type is returned in the second word of the structure provided by the user *(arg3)*.

From C, *msgenab* and *msgdisab* enable and disable message reception, respectively. Both return zero when successful. The *send, sendw, recv,* and *recvw* functions perform conditional send, unconditional send, conditional receive, and unconditional receive operations, respectively. All return the number of bytes actually sent or received, as appropriate.

SEE ALSO

—

DIAGNOSTICS

The error bit (c-bit) is set for any one of a number of error conditions. An error occurs when enabling messages if no queue is available for use; it is also erroneous to attempt to disable message reception if it is not enabled. When trying to send messages, errors occur because the message is too long, the receiver has not enabled message reception, the type specified is not valid, the receiver has an excessive number of messages outstanding on its queue, or, for conditional sends, the system message buffers are temporarily full. When receiving messages, errors may occur because the process has not enabled message reception, the requested type or size are invalid, or, for conditional receives, a message of the requested type is not on the queue. From C, a −1 return from any function indicates an error.

NICE (II) NICE (II)

NAME

nice – set program priority

SYNOPSIS

(nice = 34.)
(priority in r0)
sys nice

nice(priority)

DESCRIPTION

The scheduling *priority* of the process is changed to the argument. Positive priorities get less service than normal; 0 is default. Only the super-user may specify a negative priority. The valid range of *priority* is 20 to −220. The value of 4 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

The actual running priority of a process is the *priority* argument plus a number that ranges from 100 to 106 depending on the cpu usage of the process.

SEE ALSO

nice (I), ps(I)

DIAGNOSTICS

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user.

OPEN (II) OPEN (II)

NAME

open − open for reading or writing

SYNOPSIS

(open = 5.)
sys open; name; mode
(file descriptor in r0)

open(name, mode)
char *name;

DESCRIPTION

Open opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The returned file descriptor should be saved for subsequent calls to *read, write,* and *close.*

SEE ALSO

creat (II), read (II), write (II), close (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open. From C, a −1 value is returned on an error.

NAME

pause − suspend execution indefinitely

SYNOPSIS

(pause = 29.; not in assembler)
sys pause

pause()

DESCRIPTION

The current process is suspended from execution until a signal that is not being ignored arrives.

SEE ALSO

alarm(II)

DIAGNOSTICS

Error bit (c-bit) is set upon return. From C, a −1 is returned normally.

NAME

pipe – create an interprocess channel

SYNOPSIS

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

pipe(fildes)
int fildes[2];

DESCRIPTION

The *pipe* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor returned in r1 (resp. fildes[1]), up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned in r0 (resp. fildes[0]) will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) return an end-of-file. Write calls under similar conditions generate a fatal signal (signal (II)); if the signal is ignored, an error is returned on the write.

SEE ALSO

sh (I), read (II), write (II), fork (II)

DIAGNOSTICS

The error bit (c-bit) is set if too many files are already open. From C, a −1 returned value indicates an error. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

NAME

profil − execution time profile

SYNOPSIS

(profil = 44.; not in assembler)
sys profil; buff; bufsiz; offset; scale

profil(buff, bufsiz, offset, scale)
char buff[ ];
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz.* After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale.* If the resulting number corresponds to a word inside *buff,* that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 177777(8) gives a 1-1 mapping of pc's to words in *buff;* 77777(8) maps each pair of instruction words together. 2(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is also turned off when an *exec* is executed but remains on in child and parent both after a *fork.*

SEE ALSO

monitor (III), prof (I)

DIAGNOSTICS

—

PTRACE(II) PTRACE(II)

NAME

ptrace — process trace

SYNOPSIS

(ptrace = 26.; not in assembler)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)

ptrace(request, pid, addr, data);

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging, but it should be adaptable for simulation of non-UNIX environments. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt." See signal (II) for the list. Then the traced process enters a stopped state and its parent is notified via *wait* (II). When the child is in the stopped state, its core image can be examined and modified using *ptrace.* If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned (in r0). Request 1 indicates the data space (normally used); 2 indicates the instruction space (when I and D space are separated). *addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr,* which must be even. No useful value is returned. Request 4 specifies data space (normally used), 5 specifies instruction space. Attempts to write in pure procedure result in termination of the child, instead of going through or causing an error for the parent.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal which caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop.

8 The traced process terminates.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the ''termination'' status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec* (II) calls.

SEE ALSO

wait (II), signal (II), adb (I)

DIAGNOSTICS

From assembler, the c-bit (error bit) is set on errors; from C, −1 is returned and *errno* is set to the error code.

BUGS

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use ''illegal instruction'' signals at a very high rate) could be efficiently debugged.

Also, it should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

READ(II) READ(II)

NAME

read − read from file

SYNOPSIS

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
(nread in r0)
read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned (in r0).

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open (II), creat (II), dup (II), pipe (II)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes,* file descriptor not that of an input file. From C, a −1 return indicates the error.

SEEK(II) SEEK(II)

NAME

seek − move read/write pointer

SYNOPSIS

(seek = 19.)
(file descriptor in r0)
sys seek; offset; ptrname

seek(fildes, offset, ptrname)

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

if *ptrname* is 0, the pointer is set to *offset.*

if *ptrname* is 1, the pointer is set to its current location plus *offset.*

if *ptrname* is 2, the pointer is set to the size of the file plus *offset.*

if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

SEE ALSO

open (II), creat (II), lseek (III)

DIAGNOSTICS

The error bit (c-bit) is set for an undefined file descriptor. From C, a −1 return indicates an error.

NAME

setgid − set process group ID

SYNOPSIS

(setgid = 46.; not in assembler)
(group ID in r0)
sys setgid

setgid (gid)

DESCRIPTION

The group ID of the current process is set to the argument. Both the effective and the real group ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

SEE ALSO

getgid (II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

SETUID (II) SETUID (II)

NAME

setuid − set process user ID

SYNOPSIS

(setuid = 23.)
(user ID in r0)
sys setuid

setuid (uid)

DESCRIPTION

The user ID of the current process is set to the argument. Both the effective and the real user ID are set. This call is only permitted to the super-user or if the argument is the real user ID.

SEE ALSO

getuid (II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

SIGNAL(II) SIGNAL(II)

NAME

signal − catch or ignore signals

SYNOPSIS

(signal = 48.)
sys signal; sig; label
(old value in r0)

signal(sig, func)
int (*func)( );

DESCRIPTION

A *signal* is generated by some (typically) abnormal event, initiated either by a user at a typewriter (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals (except death of a child) cause termination of the receiving process, but this call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals:

1	hangup
2	interrupt (reboot, or delete key)
3*	quit (ascii FS)
4*	illegal instruction (not reset when caught)
5*	trace trap (not reset when caught)
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm timeout
15	catchable kill
16	unused
17	unused
18	death of a child
19	unused

In the assembler call, and for all signals except death of a child, if *label* is 0, the process is terminated when the signal occurs; this is the default action. The default action for the death of a child signal is to disregard the signal. If *label* is odd, the signal is ignored. For the death of a child signal, this means that terminated child processes are automatically removed from the system, thus eliminating the necessity of doing a *wait*(II) for terminated children. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt. Except as indicated, a signal is reset to 0 after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0, the default action for signal *sig* (termination) is reinstated. If *func* is 1, the signal is ignored. If *func* is non-zero and even, it is assumed to be the address of a function entry point. When the signal occurs, the function will be called. A return from the function will continue the process at the point it was interrupted. As in the assembler call, *signal* must in general be called again to catch subsequent signals.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write* on a slow device (like a typewriter; but not a file); and during sleep or *wait*. When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned a characteristic error status. The user's program may then, if it wishes, re-execute the call.

The starred signals in the list above cause a core image if not caught or ignored.

The value returned by the signal call is the old action defined for the signal.

After a *fork* (II) the child inherits all signals. *Exec* (II) resets all caught signals to default action but allows inheritance of ignored signals.

SEE ALSO

kill(I),kill(II),ptrace(II),wait(II),reset (III)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a −1 indicates an error; otherwise indicates success.

BUGS

SLEEP (II)

NAME

sleep − stop execution for interval

SYNOPSIS

(sleep = 35.; not in assembler)
(seconds in r0)
sys sleep

sleep(seconds)

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. This is implemented by adding the current date to the argument to produce a new date. The process is then suspended until the new date is reached.

From C, *sleep* is implemented by using signal(II) to catch the alarm clock signal, then using alarm(II) and pause(II) to set up and wait for the alarm to occur after the number of seconds specified by the argument. The treatment of the alarm clock signal by the calling program is preserved, but any alarm currently in progress is cleared.

SEE ALSO

sleep(I), alarm(II), pause(II)

DIAGNOSTICS

NAME

stat − get file status

SYNOPSIS

(stat = 18.)
sys stat; name; buf

stat(name, buf)
char *name;
struct inode *buf;

DESCRIPTION

Name points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be readable. After *stat*, *buf* has the following structure (starting offset given in bytes):

```
struct inode {
        char    minor;          /* +0: minor device of i-node */
        char    major;          /* +1: major device */
        int     inumber;        /* +2 */
        int     flags;          /* +4: see below */
        char    nlinks;         /* +6: number of links to file */
        char    uid;            /* +7: user ID of owner */
        char    gid;            /* +8: group ID of owner */
        char    size0;          /* +9: high byte of 24-bit size */
        int     size1;          /* +10: low word of 24-bit size */
        int     addr[8];        /* +12: block numbers or device number */
        int     actime[2];      /* +28: time of last access */
        int     modtime[2];     /* +32: time of last modification */
};
```

The flags are as follows:

100000		i-node is allocated
060000		2-bit file type:
	000000	plain file
	040000	directory
	020000	character-type special file
	060000	block-type special file.
010000		large file
004000		set user-ID on execution
002000		set group-ID on execution
001000		save text image after execution
000400		read (owner)
000200		write (owner)
000100		execute (owner)
000070		read, write, execute (group)
000007		read, write, execute (others)

SEE ALSO

ls (I), fstat (II), file system (V)

DIAGNOSTICS

Error bit (c-bit) is set if the file cannot be found. From C, a −1 return indicates an error.

STIME (II) STIME (II)

NAME

stime − set time

SYNOPSIS

```
(stime = 25.)
(time in r0-r1)
sys stime

stime(tbuf)
int tbuf[2];
```

DESCRIPTION

Stime sets the system's idea of the time and date. Time is measured in seconds from 0000 GMT Jan 1 1970. Only the super-user may use this call.

SEE ALSO

date (I), time (II), ctime (III)

DIAGNOSTICS

Error bit (c-bit) set if user is not the super-user.

NAME

stty − set mode of typewriter

SYNOPSIS

```
(stty = 31.)
(file descriptor in r0)
sys stty; arg
...
arg:  .byte ispeed, ospeed; 0; mode

stty(fildes, arg)
struct {
        char    ispeed, ospeed;
        int     unused;
        int     mode;
} *arg;
```

DESCRIPTION

Stty sets mode bits and character speeds for the typewriter whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the typewriter is quiescent. The input and output speeds are set from the first two bytes of the argument structure as indicated by the following table, which corresponds to the speeds supported by the DH-11 interface. If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored.

0	(hang up dataphone)
1	50 baud
2	75 baud
3	110 baud
4	134.5 baud
5	150 baud
6	200 baud
7	300 baud
8	600 baud
9	1200 baud
10	1800 baud
11	2400 baud
12	4800 baud
13	9600 baud
14	External A
15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines, in that the code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program.

The second word of the argument structure is currently unused and reserved for future use.

The *mode* contains several bits which determine the system's treatment of the typewriter:

040000	Select one of two algorithms for form-feed and vertical-tab delays
030000	Select one of four algorithms for carriage-return delays
006000	Select one of four algorithms for tab delays
001400	Select one of four algorithms for new-line delays

000200	even parity allowed on input (e. g. for M37s)
000100	odd parity allowed on input
000040	raw mode: wake up on all characters
000020	map CR into LF; echo LF or CR as CR-LF
000010	echo (full duplex)
000004	map upper case to lower on input (e. g. M33)
000002	echo and print tabs as spaces
000001	hang up (remove 'data terminal ready' lead) after last close

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is not yet defined.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 decreases with line length and is for Teletype model 40 printers at 1200 baud.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially.

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (used for GE TermiNet 300's and other terminals without the newline function).

SEE ALSO

stty (I), gtty (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a negative value indicates an error.

NAME

sync — update super-block

SYNOPSIS

(sync = 36.; not in assembler)
sys sync

DESCRIPTION

Sync causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *icheck, df,* etc. It is mandatory before a boot.

SEE ALSO

sync (VIII), update (VIII)

DIAGNOSTICS

—

NAME

tell − get file offset

SYNOPSIS

(tell = 40.; not in assembler)
(file descriptor in r0)
sys tell
(offset in r0-r1)

long tell(file)
int file;

DESCRIPTION

Tell returns the current read/write pointer associated with the open file whose descriptor is specified as the argument *file.*

SEE ALSO

seek (II)

DIAGNOSTICS

C-bit set or −1 returned for an unknown file descriptor.

BUGS

Error bit (c-bit) is set for an unknown file descriptor. From C, a −1 return indicates an error.

NAME

time – get date and time

SYNOPSIS

```
(time = 13.)
sys time

time(tvec)
int tvec[2];
```

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. From *as,* the high order word is in the r0 register and the low order is in r1. From C, the user-supplied vector is filled in.

SEE ALSO

date (I), stime (II), ctime (III)

DIAGNOSTICS

–

NAME

times – get process times

SYNOPSIS

(times = 43.; not in assembler)
sys times; buffer

times(buffer)
struct tbuffer *buffer;

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/60 seconds.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        long    proc_user_time;
        long    proc_system_time;
        long    child_user_time;
        long    child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time (I)

NAME

umount − dismount file system

SYNOPSIS

(umount = 22.)
sys umount; special

umount(special);
char *special;

DESCRIPTION

Umount announces to the system that special file *special* is no longer to contain a removable file system. A *close* is issued to the pertinent device driver. The file associated with the special file reverts to its ordinary interpretation; see *mount* (II). Only the super-user may unmount a file system.

SEE ALSO

umount (VIII), mount (II)

DIAGNOSTICS

Error bit (c-bit) set if no file system was mounted on the special file, if there are still active files on the mounted file system, or if the user is not super-user. From C, a −1 return indicates an error.

UNLINK (II) UNLINK (II)

NAME

unlink − remove directory entry

SYNOPSIS

```
(unlink = 10.)
sys  unlink; name

unlink(name)
char *name;
```

DESCRIPTION

Name points to a null-terminated string. *Unlink* removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm (I), rmdir (I), link (II)

DIAGNOSTICS

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a −1 return indicates an error.

WAIT (II)

NAME

wait – wait for process to terminate

SYNOPSIS

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

wait(status)
int *status;

DESCRIPTION

Wait causes its caller to delay until one of its child processes terminates. If any child has died since the last *wait,* return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of −1 returned). The normal return yields the process ID of the terminated child (in r0). In the case of several children several *wait* calls are needed to learn of all the deaths.

If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status*) contains the low byte of the child process r0 (resp. the argument of *exit*) when it terminated. The r1 (resp. *status*) low byte contains the termination status of the process. See signal (II) for a list of termination statuses (signals); 0 status indicates normal termination. If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit (II), fork (II), signal (II)

DIAGNOSTICS

The error bit (c-bit) is set if there are no children not previously waited for. From C, a returned value of −1 indicates an error.

NAME

write – write on a file

SYNOPSIS

(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes
(nwrite in r0)
write(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in r0). It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat (II), open (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of −1 indicates an error.

ABORT(III) ABORT(III)

NAME

abort — generate an IOT fault

SYNOPSIS

abort()

DESCRIPTION

Abort executes the IOT instruction. This is usually considered a program fault by the system and results in termination with a core dump. It is used to generate a core image for debugging.

SEE ALSO

abd (I), signal (II)

DIAGNOSTICS

usually ''IOT trap -- core dumped'' from the Shell.

BUGS

NAME

abs, fabs − absolute value

SYNOPSIS

abs (i)
int i;

double fabs (x)
double x;

DESCRIPTION

Abs returns the absolute value of its integer operand; *fabs* is the *double* version.

NAME

alloc − core allocator

SYNOPSIS

char *alloc(size)

free(ptr)
char *ptr;

DESCRIPTION

Alloc and *free* provide a simple general-purpose core management package. *Alloc* is given a size in bytes; it returns a pointer to an area at least that size which is even and hence can hold an object of any type. The argument to *free* is a pointer to an area previous allocated by *alloc;* this space is made available for further allocation.

Needless to say, grave disorder will result if the space assigned by *alloc* is overrun or if some random number is handed to *free.*

The method uses a first-fit algorithm which coalesces blocks being freed with other blocks already free. It calls *sbrk* (see *break* (I)) to get more core from the system when there is no suitable space already free. If that fails, it writes ''Out of space'' on the standard output and exits.

The external variable *slop* (which is 2 if not set) is a number such that if *n* bytes are requested, and if the first free block of size at least *n* is no larger than *n+slop,* then the whole block will be allocated instead of being split up. Larger values of *slop* tend to reduce fragmentation at the expense of unused space in the allocated blocks.

DIAGNOSTICS

''Out of space'' if it needs core and can't get it.

BUGS

NAME

atan, atan2 − arc tangent function

SYNOPSIS

```
jsr    pc,atan
jsr    pc,atan2

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

The *atan* entry returns the arc tangent of fr0 in fr0; from C, the arc tangent of *x* is returned. The range is $-\pi/2$ to $\pi/2$. The *atan2* entry returns the arc tangent of fr0/fr1 in fr0; from C, the arc tangent of *x/y* is returned. The range is $-\pi$ to π.

DIAGNOSTIC

There is no error return.

BUGS

ATOF (III) ATOF (III)

NAME

atof – convert ASCII to floating

SYNOPSIS

double atof(nptr)
char *nptr;

DESCRIPTION

Atof converts a string to a floating number. *Nptr* should point to a string containing the number; the first unrecognized character ends the number.

The only numbers recognized are: an optional minus sign followed by a string of digits optionally containing one decimal point, then followed optionally by the letter **e** followed by a signed integer.

DIAGNOSTICS

There are none; overflow results in a very large number and garbage characters terminate the scan.

BUGS

The routine should accept initial **+**, initial blanks, and **E** for **e**. Overflow should be signalled.

ATOI (III) ATOI (III)

NAME

atoi − convert ASCII to integer

SYNOPSIS

atoi(nptr)
char *nptr;

DESCRIPTION

Atoi converts the string pointed to by *nptr* to an integer. The string can contain leading blanks or tabs, an optional '−', and then an unbroken string of digits. Conversion stops at the first non-digit.

SEE ALSO

atof (III)

BUGS

There is no provision for overflow.

COMPAR (III)   COMPAR (III)

NAME

compar – default comparison routine for qsort

SYNOPSIS

```
jsr     pc,compar
```

DESCRIPTION

Compar is the default comparison routine called by *qsort* and is separated out so that the user can supply his own comparison.

The routine is called with the width (in bytes) of an element in r3 and it compares byte-by-byte the element pointed to by r0 with the element pointed to by r4.

Return is via the condition codes, which are tested by the instructions ''blt'' and ''bgt''. That is, in the absence of overflow, the condition (r0) < (r4) should leave the Z-bit off and N-bit on while (r0) > (r4) should leave Z and N off. Still another way of putting it is that for elements of length 1 the instruction

```
cmpb    (r0),(r4)
```

suffices.

Only r0 is changed by the call.

SEE ALSO

qsort (III)

BUGS

It could be recoded to run faster.

NAME

crypt − password encoding

SYNOPSIS

```
mov     $key,r0
jsr     pc,crypt

char *crypt(key)
char *key;
```

DESCRIPTION

On entry, r0 points to a string of characters terminated by an ASCII NUL. The routine performs an operation on the key which is difficult to invert (i.e. encrypts it) and leaves the resulting eight bytes of ASCII alphanumerics in a global cell called "word".

From C, the *key* argument is a string and the value returned is a pointer to the eight-character result.

This routine is used to encrypt all passwords.

SEE ALSO

passwd (I), passwd (V), login (I)

BUGS

Short or otherwise simple passwords can be decrypted easily by exhaustive search. Six characters of gibberish is reasonably safe.

NAME

ctime, localtime, gmtime — convert date and time to ASCII

SYNOPSIS

```
char *ctime(tvec)
int tvec[2];

[from Fortran]
double precision ctime
... = ctime(dummy)
int *localtime(tvec)
int tvec[2];

int *gmtime(tvec)
int tvec[2];
```

DESCRIPTION

Ctime converts a time in the vector *tvec* such as returned by time (II) into ASCII and returns a pointer to a character string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

All the fields have constant width.

The *localtime* and *gmtime* entries return pointers to integer vectors containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. The value is a pointer to an array whose components are

0	seconds
1	minutes
2	hours
3	day of the month (1-31)
4	month (0-11)
5	year − 1900
6	day of the week (Sunday = 0)
7	day of the year (0-365)
8	Daylight Saving Time flag if non-zero

The external variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, it is 5*60*60); the external variable *daylight* is non-zero if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

A routine named *ctime* is also available from Fortran. Actually it resembles more the *time* (II) system entry in that it returns the number of seconds since the epoch 0000 GMT Jan. 1, 1970 (as a floating-point number).

SEE ALSO

time(II)

BUGS

NAME

dtol − floating point to double precision integer conversion

SYNOPSIS

dtol(d,t)
double d;
int t[2];

DESCRIPTION

Dtol converts the floating point number *d* to a double precision (i.e. long) integer. The fractional part of the floating point number is lost during conversion; hence *dtol* (d,t) and *dtol* (*floor* (d),t) yield the same results.

SEE ALSO

ltod (III),floor(III)

NAME

ecvt, fcvt − output conversion

SYNOPSIS

```
jsr     pc,ecvt

jsr     pc,fcvt

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
char *fcvt(value, ndigit, decpt, sign)
...
```

DESCRIPTION

Ecvt is called with a floating point number in fr0.

On exit, the number has been converted into a string of ascii digits in a buffer pointed to by r0. The number of digits produced is controlled by a global variable *_ndigits.*

Moreover, the position of the decimal point is contained in r2: r2=0 means the d.p. is at the left hand end of the string of digits; r2>0 means the d.p. is within or to the right of the string.

The sign of the number is indicated by r1 (0 for +; 1 for −).

The low order digit has suffered decimal rounding (i. e. may have been carried into).

From C, the *value* is converted and a pointer to a null-terminated string of *ndigit* digits is returned. The position of the decimal point is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

Fcvt is identical to *ecvt,* except that the correct digit been rounded for F-style output of the number of digits specified by *ndigits.*

SEE ALSO

printf (III)

BUGS

END (III) END (III)

NAME

end, etext, edata – last locations in program

SYNOPSIS

```
extern end;
extern etext;
extern edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. Instead, their addresses coincide with the first address above the program text region *(etext),* above the initialized data region *(edata),* or uninitialized data region *(end).* The last is the same as the program break. Values are given to these symbols by the link editor *ld* (I) when, and only when, they are referred to but not defined in the set of programs loaded.

The usage of these symbols is rather specialized, but one plausible possibility is

```
extern end;
...
... = brk(&end+...);
```

(see *break* (II)). The problem with this is that it ignores any other subroutines which may want to extend core for their purposes; these include *sbrk* (see *break* (II)), *alloc* (III), and also secret subroutines invoked by the profile (−p) option of *cc*. Of course it was for the benefit of such systems that the symbols were invented, and user programs, unless they are in firm control of their environment, are wise not to refer to the absolute symbols directly.

One technique sometimes useful is to call *sbrk(0),* which returns the value of the current program break, instead of referring to *&end,* which yields the program break at the instant execution started.

These symbols are accessible from assembly language if it is remembered that they should be prefixed by '_'

SEE ALSO

break (II), alloc (III)

BUGS

NAME

exp − exponential function

SYNOPSIS

```
jsr     pc,exp
```

```
double exp(x)
double x;
```

DESCRIPTION

The exponential of fr0 is returned in fr0. From C, the exponential of *x* is returned.

DIAGNOSTICS

If the result is not representable, the c-bit is set and the largest positive number is returned. From C, no diagnostic is available.

Zero is returned if the result would underflow.

BUGS

FLOOR (III)

NAME

floor, ceil − floor and ceiling functions

SYNOPSIS

```
double floor(x)
double x;

double ceil(x)
double x;
```

DESCRIPTION

The floor function returns the largest integer (as a double precision number) not greater than **x**.

The ceil function returns the smallest integer not less than **x**.

BUGS

FMOD (III) FMOD (III)

NAME

fmod − floating modulo function

SYNOPSIS

```
double fmod(x, y)
double x, y;
```

DESCRIPTION

Fmod returns the number f such that $x = iy + f$, i is an integer, and $0 \leqslant f < y$.

BUGS

FPTRAP (III) FPTRAP (III)

NAME

fptrap − floating point interpreter

SYNOPSIS

sys signal; 4; fptrap

DESCRIPTION

Fptrap is a simulator of the 11/45 FP11-B floating point unit. It works by intercepting illegal instruction traps and decoding and executing the floating point operation codes.

FILES

In systems with real floating point, there is a fake routine in /lib/liba.a with this name; when simulation is desired, the real version should be put in liba.a.

DIAGNOSTICS

A break point trap is given when a real illegal instruction trap occurs.

SEE ALSO

signal (II), cc (I) ('−f' option)

BUGS

Rounding mode is not interpreted. It's slow.

NAME

gamma − log gamma function

SYNOPSIS

```
jsr     pc,gamma

double gamma(x)
double x;
```

DESCRIPTION

If *x* is passed (in fr0) *gamma* returns $\ln |\Gamma(x)|$ (in fr0). The sign of $\Gamma(x)$ is returned in the external integer *signgam.* The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

DIAGNOSTICS

The c-bit is set on negative integral arguments and the maximum value is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

getarg, iargc – get command arguments from Fortran

SYNOPSIS

call getarg (i, iarray [, isize])

... = iargc(dummy)

DESCRIPTION

The *getarg* entry fills in *iarray* (which is considered to be *integer)* with the Hollerith string representing the *i* th argument to the command in which it it is called. If no *isize* argument is specified, at least one blank is placed after the argument, and the last word affected is blank padded. The user should make sure that the array is big enough.

If the *isize* argument is given, the argument will be followed by blanks to fill up *isize* words, but even if the argument is long no more than that many words will be filled in.

The blank-padded array is suitable for use as an argument to setfil (III).

The *iargc* entry returns the number of arguments to the command, counting the first (file-name) argument.

SEE ALSO

exec (II), setfil (III)

BUGS

GETC (III) GETC (III)

NAME

getc, getw, fopen — buffered input

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fopen; iobuf

fopen(filename, iobuf)
char *filename;
struct buf *iobuf;
jsr     r5,getc; iobuf
(character in r0)
getc(iobuf)
struct buf *iobuf;

jsr     r5,getw; iobuf
(word in r0)

getw(iobuf)
struct buf *iobuf;
```

DESCRIPTION

These routines provide a buffered input facility. *Iobuf* is the address of a 518(10) byte buffer area whose contents are maintained by these routines. Its structure is

```
struct buf {
        int fildes;     /* File descriptor */
        int nleft;      /* Chars left in buffer */
        char *nextp;    /* Ptr to next character */
        char buff[512];         /* The buffer */
};
```

Fopen may be called initially to open the file. On return, the error bit (c-bit) is set if the open failed. If *fopen* is never called, *get* will read from the standard input file. From C, the value is negative if the open failed.

Getc returns the next byte from the file in r0. The error bit is set on end of file or a read error. From C, the character is returned as an integer, without sign extension; it is −1 on end-of-file or error.

Getw returns the next word in r0. *Getc* and *getw* may be used alternately; there are no odd/even problems. *Getw* may be called from C; −1 is returned on end-of-file or error, but of course is also a legitimate value.

Iobuf must be provided by the user; it must be on a word boundary.

To reuse the same buffer for another file, it is sufficient to close the original file and call *fopen* again.

SEE ALSO

open (II), read (II), getchar (III), putc (III)

DIAGNOSTICS

c-bit set on EOF or error; from C, negative return indicates error or EOF. Moreover, *errno* is set by this routine just as it is for a system call (see introduction (II)).

BUGS

GETCHAR (III)

NAME

getchar − read character

SYNOPSIS

getchar( )

DESCRIPTION

Getchar provides the simplest means of reading characters from the standard input for C programs. It returns successive characters until end-of-file, when it returns "\0"

Associated with this routine is an external variable called *fin*, which is a structure containing a buffer such as described under *getc* (III).

Generally speaking, *getchar* should be used only for the simplest applications; *getc* is better when there are multiple input files.

SEE ALSO

getc (III)

DIAGNOSTICS

Null character returned on EOF or error.

BUGS

−1 should be returned on EOF; null is a legitimate character.

NAME

getpw − get name from UID

SYNOPSIS

getpw (uid, buf)
char *buf;

DESCRIPTION

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

passwd (V)

DIAGNOSTICS

non-zero return on error.

BUGS

NAME

hmul − high-order product

SYNOPSIS

hmul(x, y)

DESCRIPTION

Hmul returns the high-order 16 bits of the product of **x** and **y**. (The binary multiplication operator generates the low-order 16 bits of a product.)

BUGS

NAME

hypot − calculate hypotenuse

SYNOPSIS

jsr r5,hypot

DESCRIPTION

The square root of fr0 × fr0 + fr1 × fr1 is returned in fr0. The calculation is done in such a way that overflow will not occur unless the answer is not representable in floating point.

DIAGNOSTICS

The c-bit is set if the result cannot be represented.

BUGS

NAME

ierror — catch Fortran errors

SYNOPSIS

if (ierror (*errno* **) .ne. 0) goto** *label*

DESCRIPTION

Ierror provides a way of detecting errors during the running of a Fortran program. Its argument is a run-time error number such as enumerated in *fc* (I).

When *ierror* is called, it returns a 0 value; thus the **goto** statement in the synopsis is not executed. However, the routine stores inside itself the call point and invocation level. If and when the indicated error occurs, a **return** is simulated from *ierror* with a non-zero value; thus the **goto** (or other statement) is executed. It is a ghastly error to call *ierror* from a subroutine which has already returned when the error occurs.

This routine is essentially tailored to catching end-of-file situations. Typically it is called just before the start of the loop which reads the input file, and the **goto** jumps to a graceful termination of the program.

There is a limit of 5 on the number of different error numbers which can be caught.

SEE ALSO

fc (I)

BUGS

There is no way to ignore errors.

ITOL(III) ITOL(III)

NAME

itol − integer to long integer conversion

SYNOPSIS

long itol(hi, lo)
int hi, lo;

DESCRIPTION

Itol combines the two integers *hi* and *lo* to form a long integer. This allows integers to be converted to long integers without sign extension.

SEE ALSO

ltoi (III)

NAME

ldiv, lrem − long division

SYNOPSIS

ldiv(hidividend, lodividend, divisor)

lrem(hidividend, lodividend, divisor)

DESCRIPTION

The concatenation of the signed 16-bit *hidividend* and the unsigned 16-bit *lodividend* is divided by *divisor.* The 16-bit signed quotient is returned by *ldiv* and the 16-bit signed remainder is returned by *lrem.* Divide check and erroneous results will occur unless the magnitude of the quotient is less than 32768.

An integer division of an unsigned dividend by a signed divisor may be accomplished by

```
quo = ldiv(0, dividend, divisor);
```

and similarly for the remainder operation.

Often both the quotient and the remainder are wanted. Therefore *ldiv* leaves a remainder in the external cell *ldivr.*

BUGS

No check on divide check.

LNXX(III)

NAME

lnxx — return name of current terminal

SYNOPSIS

jsr pc,lnff
(filedes in r0)

jsr pc,lnxx
(standard input assumed)

lnxx(filedes)

DESCRIPTION

Lnxx extracts the last character (or 2 characters for double character identifiers) of the terminal name which is the standard input or is specified by the argument *filedes* (file descriptor of a file). If the terminal name is "/dev/ln*k*", *k* is returned; if "/dev/ln*kk*", *kk* is returned.

If the file description does not correspond to a terminal, "**xx**" is returned.

BUGS

NAME

locv — long output conversion

SYNOPSIS

char *locv(hi, lo)
int hi, lo;

DESCRIPTION

Locv converts a signed double-precision integer, whose parts are passed as arguments, to the equivalent ASCII character string and returns a pointer to that string.

SEE ALSO

atof (III), atoi (III)

BUGS

Since *locv* returns a pointer to a static buffer containing the converted result, it cannot be used twice in the same expression; the second result overwrites the first.

NAME

log − natural logarithm

SYNOPSIS

```
jsr     pc,log
```

```
double log(x)
double x;
```

DESCRIPTION

The natural logarithm of fr0 is returned in fr0. From C, the natural logarithm of **x** is returned.

DIAGNOSTICS

The error bit (c-bit) is set if the input argument is less than or equal to zero and the result is a negative number very large in magnitude. From C, there is no error indication.

BUGS

LSEEK(III) LSEEK(III)

NAME

lseek −seek using a long offset

SYNOPSIS

lseek(fildes,lng-offset,ptrname)
long lng-offset;

DESCRIPTION

Lseek performs the appropriate combination of block and character *seeks* to position the read/write pointer of a file based on the long offset argument. *Ptrname* has the same meaning as in *seek* except only the values 0, 1 and 2 are legal.

SEE ALSO

seek (II)

DIAGNOSTICS

A −1 is returned if *ptrname* is greater than 3 or a seek error occurred.

NAME

ltod – double precision integer to floating point conversion

SYNOPSIS

```
double ltod(t)
int       t[2];
```

DESCRIPTION

Ltod converts a signed double precision (i.e.long) integer to the equivalent floating point number.

SEE ALSO

dtol (III)

LTOI(III) LTOI(III)

NAME

ltoi — long integer to integer conversion

SYNOPSIS

ltoi(lng)
long lng;

DESCRIPTION

Ltoi converts a long integer to a single precision integer by truncating the high order word.

SEE ALSO

itol (III)

NAME

mesg – write message on typewriter

SYNOPSIS

jsr r5,mesg; <Now is the time\0>; .even

DESCRIPTION

Mesg writes the string immediately following its call onto the standard output file. The string must be terminated by an ASCII NULL byte.

BUGS

MKTEMP (III)

NAME

mktemp — make a unique named temporary file

SYNOPSIS

mktemp(as)
char *as;

DESCRIPTION

Mktemp creates a unique named temporary file. The name is made up of the string pointed to by *as* where each 'X' is replaced by a digit formed from the process identification. If the named file already exists the letters 'a', 'b', ... are added as the last letter and the name is tried again.

Mktemp returns a pointer to the unique name. If all the letters of the alphabet are exhausted without successfully creating a unique name the name '/' is returned.

BUGS

MONITOR (III) MONITOR (III)

NAME

monitor – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize)
int lowpc( ), highpc( ), buffer[ ], bufsize;

DESCRIPTION

Monitor is an interface to the system's *profile* (II) entry. *Lowpc* and *highpc* are the names of two functions; *buffer* is the address of a (user supplied) array of *bufsize* integers. *Monitor* arranges for the system to sample the user's program counter periodically and record the execution histogram in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor(2, &etext, buf, bufsize);
```

Etext is a loader-defined symbol which lies just above all the program text.

To stop execution monitoring and write the results on the file *mon.out,* use

```
monitor(0);
```

Then, when the program exits, prof (I) can be used to examine the results.

It is seldom necessary to call this routine directly; the **−p** option of *cc* is simpler if one is satisfied with its default profile range and resolution.

FILES

mon.out

SEE ALSO

prof (I), profil (II), cc (I)

NARGS (III)

NAME

nargs − argument count

SYNOPSIS

nargs( )

DESCRIPTION

Nargs returns the number of actual parameters supplied by the caller of the routine which calls *nargs.*

The argument count is accurate only when none of the actual parameters is *float* or *double.* Such parameters count as four arguments instead of one.

BUGS

As indicated. This routine does not work (and cannot be made to work) in programs with separated I and D space. Altogether it is best to avoid using this routine and depend, for example, on passing an explicit argument count.

NLIST (III)

NAME

nlist − get entries from name list

SYNOPSIS

```
nlist(filename, nl)
char *filename;
struct {
        char    name[8];
        int     type;
        int     value;
} nl[ ];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of a list of 8-character names (null padded) each followed by two words. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are placed in the two words following the name. If the name is not found, the type entry is set to −1.

This subroutine is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out (V)

DIAGNOSTICS

All type entries are set to −1 if the file cannot be found or if it is not a valid namelist.

BUGS

NAME

perror, sys_errlist, sys_nerr, errno − system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;
```

DESCRIPTION

Perror produces a short error message describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno,* which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

Introduction to System Calls

BUGS

NAME

pow − floating exponentiation

SYNOPSIS

```
movf   x,fr0
movf   y,fr1
jsr    pc,pow

double pow(x,y)
double x, y;
```

DESCRIPTION

Pow returns the value of x^y (in fr0). *Pow(0.0, y)* is 0 for any *y*. *Pow(−x, y)* returns a result only if *y* is an integer.

SEE ALSO

exp (III), log (III)

DIAGNOSTICS

The carry bit is set on return in case of overflow, *pow(0.0, 0.0)*, or *pow(−x, y)* for non-integral *y*. From C there is no diagnostic.

BUGS

NAME

printf – formatted print

SYNOPSIS

printf(format, arg_1, ...);
char *format;

DESCRIPTION

Printf converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf.*

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign "–" which specifies *left adjustment* of the converted argument in the indicated field;
- an optional digit string specifying a *field width;* if the converted argument has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width;
- an optional period "." which serves to separate the field width from the next digit string;
- an optional digit string *(precision)* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are

d o x	The integer argument is converted to decimal, octal, or hexadecimal notation respectively.
D O X	The long integer argument is converted to decimal, octal, or hexadecimal notation respectively.
f	The argument is converted to decimal notation in the style "[–]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. The argument should be *float* or *double.*
e	The argument is converted in the style "[–]d.ddde±dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The argument should be a *float* or *double* quantity.
c	The argument character is printed.
s	The argument is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by

the precision specification is reached; however if the precision is 0 or missing, all characters up to a null are printed.

u
l The argument is taken to be an unsigned integer which is converted to decimal and printed (the result will be in the range 0 to 65535).

If no recognizable character appears after the %, that character is printed; thus % may be printed by use of the string %%. In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar*.

SEE ALSO

putchar (III)

BUGS

Very wide fields (>128 characters) fail.

PUTC (III)

NAME

putc, putw, fcreat, fflush — buffered output

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fcreat; iobuf

fcreat(file, iobuf)
char *file;
struct buf *iobuf;

(put byte in r0)
jsr     r5,putc; iobuf

putc(c, iobuf)
int c;
struct buf *iobuf;

(put word in r0)
jsr     r5,putw; iobuf

putw(w, iobuf);
int w;
struct buf *iobuf;

jsr     r5,flush; iobuf

fflush(iobuf)
struct buf *iobuf;
```

DESCRIPTION

Fcreat creates the given file (mode 666) and sets up the buffer *iobuf* (size 518 bytes); *putc* and *putw* write a byte or word respectively onto the file; *flush* forces the contents of the buffer to be written, but does not close the file. The structure of the buffer is:

```
struct buf {
        int fildes;     /* File descriptor */
        int nunused;    /* Remaining slots */
        char *xfree;    /* Ptr to next free slot */
        char buff[512];         /* The buffer */
};
```

Before terminating, a program should call *flush* to force out the last of the output (*fflush* from C).

The user must supply *iobuf,* which should begin on a word boundary.

To write a new file using the same buffer, it suffices to call *[f]flush,* close the file, and call *fcreat* again.

SEE ALSO

creat (II), write (II), getc (III)

DIAGNOSTICS

Fcreat sets the error bit (c-bit) if the file creation failed (from C, returns −1). *Putc* and *putw* return their character (word) argument. In all calls *errno* is set appropriately to 0 or to a system error number. See *intro* (II).

BUGS

PUTCHAR (III) PUTCHAR (III)

NAME

putchar, flush − write character

SYNOPSIS

putchar(ch)

flush( )

DESCRIPTION

Putchar writes out its argument and returns it unchanged. Only the low-order byte is written, and only if it is non-null. Unless other arrangements have been made, *putchar* writes in unbuffered fashion on the standard output file.

Associated with this routine is an external variable *fout* which has the structure of a buffer discussed under *putc* (III). If the file descriptor part of this structure (first word) is greater than 2, output via *putchar* is buffered. To achieve buffered output one may say, for example:

```
fout = dup(1);                    or
fout = creat(...);
```

In such a case *flush* must be called before the program terminates in order to flush out the buffered output. *Flush* may be called at any time.

SEE ALSO

putc (III)

BUGS

The *fout* notion is questionable.

NAME

qsort − quicker sort

SYNOPSIS

qsort(base, nel, width, compar)
char *base;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according to whether the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort (I)

BUGS

NAME

rand, srand – random number generator

SYNOPSIS

```
(seed in r0)
jsr     pc,srand        /to initialize

jsr     pc,rand         /to get a random number

srand(seed)
int seed;

rand( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator to return successive pseudo-random numbers (in r0) in the range from 0 to $2^{15}-1$.

The generator is reinitialized by calling *srand* with 1 as argument (in r0). It can be set to a random starting point by calling *srand* with whatever you like as argument, for example the low-order word of the time.

BUGS

The low-order bits are not very random.

NAME

reset, setexit − execute non-local goto

SYNOPSIS

setexit( )

reset( )

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setexit saves its stack environment in a static place for later use by *reset.*

Reset restores the environment saved by the last call of *setexit.* It then returns in such a way that execution continues as if the call of *setexit* had just returned. All accessible data have values as of the time *reset* was called.

The routine that called *setexit* must still be active when *reset* is called.

SEE ALSO

signal (II)

BUGS

NAME

setfil – specify Fortran file name

SYNOPSIS

call setfil (unit, hollerith-string)

DESCRIPTION

Setfil provides a primitive way to associate an integer I/O *unit* number with a file named by the *hollerith-string.* The end of the file name is indicated by a blank. Subsequent I/O on this unit number will refer to the file whose name is specified by the string.

Setfil should be called only before any I/O has been done on the *unit,* or just after doing a **rewind** or **endfile.** It is ineffective for unit numbers 5 and 6.

SEE ALSO

fc (I)

BUGS

The exclusion of units 5 and 6 is unwarranted.

SIN (III) SIN (III)

NAME

sin, cos − trigonometric functions

SYNOPSIS

```
jsr     pc,sin (cos)

double sin(x)
double x;

double cos(x)
double x;
```

DESCRIPTION

The sine (cosine) of fr0 (resp. **x**), measured in radians, is returned (in fr0).

The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

BUGS

NAME

sqrt − square root function

SYNOPSIS

jsr pc,sqrt

double sqrt(x)
double x;

DESCRIPTION

The square root of fr0 (resp. **x**) is returned (in fr0).

DIAGNOSTICS

The c-bit is set on negative arguments and 0 is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

cm – Minitape interface

DESCRIPTION

This driver is a character device, and supports up to four Minitape drives. The four track cartridge used by the Minitape system is treated as either four separate tapes or as a single tape. The minor device number determines the drive and track treatment as follows:

The drive # is carried by bits 4,3.
Track usage is carried by bits 2,1,0

track 1 only, rewound	0 0 0
track 2 only, rewound	0 0 1
track 3 only, rewound	0 1 0
track 4 only, rewound	0 1 1
multi-track rewound	1 0 0
multi-track no-rewind	1 0 1

Only the multi-track no-rewind mode can be used for multi-file operations. All modes will accept record lengths of from 10 to 2048 bytes . When a write operation is closed a double end-of-file is written. A zero is returned when a tape mark is detected. A -1 is returned when an error or final EOT is encountered, and the status of the drive is stored in the driver.

FILES

SEE ALSO

BUGS

Before starting a multi-file operation the cartridge to be used must be initialized via a user supplied program which opens the drive for reading on track 1 and then exits.

NAME

dc − DC-11 communications interface

DESCRIPTION

The discussion of typewriter I/O given in *tty* (IV) applies to these devices.

The DC-11 typewriter interface operates at any of four speeds, independently settable for input and output. The speed is selected by the same encoding used by the *dh* (IV) device (enumerated in *stty* (II)); impossible speed changes are ignored.

FILES

/dev/ln??

SEE ALSO

tty (IV), stty (II), dh (IV), gtty (II)

BUGS

Although impossible speed changes are not attempted on the hardware, the software is quite willing to accept them and alter the software state of the device accordingly. Subsequent requests for the device's state (*gtty* (II)) provide an erroneous status.

NAME

dh – DH-11 communications multiplexer

DESCRIPTION

Each line attached to the DH-11 communications multiplexer behaves as described in *tty* (IV). Input and output for each line may independently be set to run at any of 16 speeds; see *stty* (II) for the encoding.

FILES

/dev/ln??

SEE ALSO

tty (IV), stty (II)

BUGS

NAME

dn – DN-11 ACU interface

DESCRIPTION

The *dn?* files are write-only. The permissible codes are:

0-9	dial 0-9
–	8 second delay for second dial tone

The entire telephone number must be presented in a single *write* system call.

FILES

/dev/dn?

SEE ALSO

dp (IV)

BUGS

NAME

dp – DP-11 201 data-phone interface

DESCRIPTION

The */dev/dp0* file is a 201 data-phone interface. *Read*(II) and *write*(II) calls to this file are limited to a maximum of 512 bytes. Each write call is sent as a single record. Seven bits from each byte are written along with an eighth odd parity bit. The sync must be user supplied. Each read call returns characters received from a single record. Seven bits are returned unaltered; the eighth bit is set if the byte was not received in odd parity. A 10-second time-out interval is used; a zero-byte record is returned if nothing is received within 10 seconds of a read.

FILES

/dev/dp0

SEE ALSO

dn (IV)

BUGS

HD(IV) HD(IV)

NAME

hd - half-duplex terminal interface

DESCRIPTION

This section describes the half-duplex protocol implemented as a line discipline which permits access to UNIX using Teletype Model 40/1's and Model 40/2's (in 202 mode) using 202S data sets or null modems. This line discipline also supports the Tektronix 4010 and 4012 when equipped with the optional communications interface. Only the aspects of the half-duplex interface that are different from the general tty interface as discussed in tty(IV) will be covered.

When a special file associated with the half-duplex protocol is opened, the interface mode is 1200 baud; either parity accepted; 10 bits/character; and CR action character. The protocol is designed to support half-duplex terminals (such as the Teletype Model 40/1) using half-duplex communications links such as those provided by 202S data sets. This prevents simultaneous send and receive usually available when using UNIX. However, you may send to UNIX anytime UNIX is not sending to you. The output from UNIX may be temporarily suspended by sending a reverse channel break. This is done when using a Model 40/1 or 40/2 by depressing the *INTRPT* key twice; the first time will light the key top light (dropping reverse channel) the second time will turn the light off (raising reverse channel). If reverse channel remains off for more than about 6 seconds UNIX will hang up. The output once stopped may be continued by a repeat of the above procedure or an ASCII DEL or FS character may be entered with its usual affect (see tty(IV)).

The Teletype Model 40/1 can transmit either a line-at-a-time or a block-at-a-time. When the carriage return is depressed, all characters from the beginning of the current line up to and including the carriage return are transmitted. The cursor is left at the end of the line sent. To avoid sending the prompt sequence, "% ", back to UNIX with your next line of input and to advance to the next line UNIX sends a newline when it expects input from the terminal. At any time any number of lines (<= the capacity of the terminal being used) may be sent at once by using the Model 40 block mode; clist overflow is prevented by dropping reverse channel to temporally halt input from the terminal. Input is resumed when clist use has dropped to a reasonable level again. ASCII GS and DC4 are recognized as special and may be used as block ending characters.

NAME

hp − RH-11/RP04 moving-head disk

DESCRIPTION

The files *hp0, ..., hp31* refer to sections of RP disk drive 0. The files *hp32, ..., hp63* refer to drive 1 etc. This is done since the size of a full RP drive is larger than the maximum file system size. This allows a disk to be broken up into more manageable pieces. The driver recognizes and maps the hp identification into disk sections as defined below.

hp#	start	length	cylinder
0	0	9614	0 thru 22
			23 thru 43 possible swap area
1	18392	65535	44 thru 200
2	84018	65535	201 thru 357
3	149644	20900	358 thru 407
			408 thru 410 spare
4	0	40600	0 thru 97
5	41800	40600	100 thru 197
6	84018	40600	201 thru 298
7	125400	40600	300 thru 397
8	0	18300	0 thru43
9	18392	65535	44 thru 201
10	84436	65535	202 thru 359
11	150480	20000	360 thru 407
12	170544	65535	408 thru 565
13	236588	65535	566 thru 723
14	302632	37200	724 thru 812
			813 thru 814 spare
15	18392	33000	44 thru 122
16	51414	33000	123 thru 201
17	84436	33000	202 thru 280
18	117458	33000	281 thru 359
			360 thru 379 possible swap area
19	158840	11700	380 thru 407
20	170544	33000	408 thru 486
21	203566	33000	487 thru 565
22	236588	33000	566 thru 644
23	269610	33000	645 thru 723
24−31			user definable

The sectioning for additional drives is similar. RP04, RP05 and RP06 disk drives are supported. Cylinders numbered greater than 410 are for use on RP06 drives only.

Users must configure their disks using a consistent set of these sections. Use of overlapping sections is risky.

The *hp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rhp* and end with a number which selects the same disk section as the corresponding *hp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hp?, /dev/rhp?

NAME

hs – RH11/RS03-RS04 fixed-head disk file

DESCRIPTION

The files *hs0*, ..., *hs7* refer to RJS03 disk drives 0 through 7. The files *hs8*, ..., *hs15* refer to RJS04 disk drives 0 through 7. The RJS03 drives are each 1024 blocks long and the RJS04 drives are 2048 blocks long.

The *hs* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" inteface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw HS files begin with *rhs*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hs?, /dev/rhs?

BUGS

HT (IV)

NAME

ht − RH-11/TU-16 magtape interface

DESCRIPTION

The files *mt?* refer to the DEC RH/TM/TU16 magtape and have the following meaning:

mt0, ..., mt3	800 bpi, rewind
mt4, ..., mt7	800 bpi, no rewind
mt8, ..., mt11	1600 bpi, rewind
mt12, ..., mt15	1600 bpi, no rewind

When a rewind *mt* file is closed, the tape is rewound; if it was open for writing, a double end-of-file is written first. Conversely, the tape is not rewound for a no-rewind *mt* file; a single end-of-file is written if the file was open for writing. By judiciously choosing *mt* files, it is possible to handle multi-file tapes.

A standard tape consists of a series of 512-byte records terminated by a double end-of-file. To the extent possible (even though it is inefficient), the system allows the tape to be treated like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the ‘‘raw’’ interface is appropriate. The associated files are named *rmt0, ..., rmt7* and *rmt8, ..., rmt15.* Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

tp (I), mtm (I)

BUGS

The magtape system is supposed to be able to take 64 drives. Such addressing has never been tried. In fact, with the mapping above, four drives are the most ever supported.

If any non-data error is encountered, it refuses to do anything more until closed.

NAME

kl – KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of typewriter I/O given in *tty* (IV) applies to these devices.

Since they run at a constant speed, attempts to change the speed via *stty* (II) are ignored.

The on-line console typewriter is interfaced using a KL-11 or DL-11. By appropriate switch settings during a reboot, UNIX will come up as a single-user system with I/O on the console typewriter.

FILES

/dev/ln00	console

SEE ALSO

tty (IV), init (VIII)

BUGS

NAME

lp − line printer

DESCRIPTION

Lp provides the interface to any of the standard DEC line printers. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII (64-character) mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

{	(-
}	)-
`	'-
|	!-
~	−

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. A sequence of newlines which extends over the end of a page is turned into a form feed. Lines longer than 80 characters are truncated; the page image can be moved 8 spaces to the right. Both of these numbers are parameters in the driver.

FILES

/dev/lp

SEE ALSO

lpr (I)

BUGS

Half-ASCII mode and the maximum line length should be set-able by a call analogous to *stty* (II).

If the first characters written are newlines, they are thrown away.

NAME

pc — PC-11 paper tape reader/punch

DESCRIPTION

Ppt refers to the PC-11 paper tape reader or punch, depending on whether it is read or written.

When *ppt* is opened for writing, a 100-character leader is punched. Thereafter each byte written is punched on the tape. No editing of the characters is performed. When the file is closed, a 100-character trailer is punched.

When *ppt* is opened for reading, the process waits until tape is placed in the reader and the reader is on-line. Then requests to read cause the characters read to be passed back to the program, again without any editing. This means that several null leader characters will usually appear at the beginning of the file. Likewise several nulls are likely to appear at the end. End-of-file is generated when the tape runs out.

Seek calls for this file are meaningless.

FILES

/dev/ppt

BUGS

If both the reader and the punch are open simultaneously, the trailer is sometimes not punched. Sometimes the reader goes into a dead state in which it cannot be opened.

MEM (IV) MEM (IV)

NAME

mem, kmem, null – core memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system using the debugger.

A memory address is an 18-bit quantity which is used directly as a UNIBUS address. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed. In particular, the I/O area of *kmem* is located beginning at 160000 (octal) rather than at 760000. The 1K region beginning at 140000 (octal) is the system's data for the current process. It should be kept in mind that in systems with separate I (Instruction) and D (Data) space, the mapping for the file *kmem* is done through the kernel's D space registers.

The file *null* returns end-of-file on *read* and ignores *write.*

FILES

/dev/mem, /dev/kmem, /dev/null

RF (IV)　　　RF (IV)

NAME

rf − RF11/RS11 fixed-head disk file

DESCRIPTION

This file refers to the concatenation of all RS-11 disks.

Each disk contains 1024 256-word blocks. The length of the combined RF file is 1024×(minor+1) blocks.

The *rf0* file accesses the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The name of the raw RF file is *rrf0*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rf0, /dev/rrf0

BUGS

The 512-byte restrictions on the raw device are not physically necessary, but are still imposed.

NAME

rk – RK-11/RK03 (or RK05) disk

DESCRIPTION

Rk? refers to an entire RK03 disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871.

Drive numbers (minor devices) of eight and larger are treated specially. Drive 8+*x* is the *x*+1 way of interleaving devices rk0 to rk*x*. Thus blocks on rk10 are distributed alternately among rk0, rk1, and rk2.

The *rk* files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with *rrk* and end with a number which selects the same disk as the corresponding *rk* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rk?, /dev/rrk?

BUGS

Care should be taken in using the interleaved files. First, the same drive should not be accessed simultaneously using the ordinary name and as part of an interleaved file, because the same physical blocks have in effect two different names; this fools the system's buffering strategy. Second, the combined files cannot be used for swapping or raw I/O.

RP (IV) RP (IV)

NAME

rp — RP-11/RP03 moving-head disk

DESCRIPTION

The files *rp0, ..., rp7* refer to sections of RP disk drive 0; the files *rp8, ..., rp15* refer to drive 1; etc. This is done since the size of a full RP drive is 81200 blocks and internally the system is only capable of addressing 65536 blocks. Since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the pseudo-disks on each drive are as follows:

disk	start	length	cylinder
0	0	40600	0 thru 202
1	40600	40600	203 thru 405
2	0	9200	0 thru 45
3	72000	9200	360 thru 405
4	0	65535	0 thru 327
5	15600	65535	78 thru 405
6-7	unassigned		

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. Here is a suggestion for two useful configurations: If the root of the file system is on some other device and the RP used as a mounted device, then *rp0* and *rp1,* which divide the disk into two equal size portions, is a good idea. Other things being equal, it is advantageous to have two equal-sized portions since one can always be copied onto the other, which is occasionally useful.

If the RP is the only disk and has to contain the root and the swap area, the root can be put on *rp2* and a mountable file system on *rp5.* Then the swap space can be put in the unused blocks 9200 to 15600 of *rp0* (or, equivalently, *rp4).* This arrangement puts the root file system, the swap area, and the i-list of the mounted file system relatively near each other and thus tends to minimize head movement.

The *rp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rrp* and end with a number which selects the same disk section as the corresponding *rp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rp?, /dev/rrp?

BUGS

NAME

tc − TC-11/TU56 DECtape

DESCRIPTION

The files *tap0, ..., tap7* refer to the TC-11/TU56 DECtape drives 0 to 7.

The 256-word blocks on a standard DECtape are numbered 0 to 577.

FILES

/dev/tap?

SEE ALSO

tp (I)

BUGS

NAME

tm − TM-11/TU-10 magtape interface

DESCRIPTION

The files *mt0, ..., mt7* refer to the DEC TU10/TM11 magtape. When one of *mt0, ..., mt3* is closed, the tape is rewound; if it was open for writing, a double end-of-file is written first. Conversely, when one of *mt4, ..., mt7* is closed, the tape is not rewound; a single end-of-file is written if the tape was open for writing. By judiciously choosing *mt* files, it is possible to read and write multi-file tapes.

A standard tape consists of a series of 512-byte records terminated by an end-of-file. To the extent possible (even though it is inefficient), the system allows the tape to be treated like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the ''raw'' interface is appropriate. The associated files are named *rmt0, ..., rmt7*. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

tp (I), mtm (I)

BUGS

If any non-data error is encountered, it refuses to do anything more until closed.

With the above mapping there is no way of getting to physical drives 4 to 7.

TTY(IV) TTY(IV)

NAME

tty − interface to low speed asynchronous devices including typewriters

DESCRIPTION

This section describes the general nature of the interface to low speed asynchrounous devices including the character processing that takes place in UNIX interaction with typewriters.

The low speed asynchronous devices all use the same general software interface, no matter what hardware is involved. The *kl, dc* and *dh* writeups (IV) describe peculiarities of the individual devices; the remainder of this section describes common features of the overall interface.

The devices discussed here differ from other UNIX devices in that alternate control strategies, referred to as *line disciplines,* are permitted. Line discipline *0* is the control strategy used for the familiar UNIX interaction with typewriters and is described below.

Typewriter Interface

When a typewriter (line discipline *0)* file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *getty* and become a user's input and output file. The very first process to open a typewriter file defines the process group with which the typewriter is associated. The process group plays a special role in handling quit and interrupt signals, as discussed below. The process group is inherited by a child process during a fork.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters which have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

There are a number of modes which can be changed by use of the *stty* system call (II). When first opened, the interface mode is 300 baud; either parity accepted; character echoing and newline as an end of line delimiter. Modes that can be changed by *stty* include the interface speed (if the hardware permits); acceptance of even parity, odd parity, or both; a raw mode in which all characters may be read one at a time; a carriage return (CR) mode in which CR is mapped into newline on input and either CR or line feed (LF) cause echoing of the sequence LF-CR; mapping of upper case letters into lower case; suppression of echoing; a variety of delays after function characters; and the printing of tabs as spaces. See *getty* (VIII) for the way that terminal speed and type are detected.

Normally, typewriter input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. The character '#' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. The character '@' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either '@' or '#' may be entered literally by preceding it by '\'; the erase or kill character remains, but the '\' disappears. Substitutes for '#' and '@' as erase and kill characters can be specified (see below).

When desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by ‘\’. In addition, the following escape sequences are generated on output and accepted on input:

for	use
`	\′
|	\!
~	\
{	\(
}	\)

In raw mode, the program reading is awakened on each character. No erase or kill processing is done; and the EOT, quit and interrupt characters are not treated specially. The input parity bit is passed back to the reader, but parity is still generated for output characters.

The ASCII EOT (control-D) character may be used to generate an end of file from a typewriter. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication. The EOT is passed back unchanged in raw mode.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a *hangup* signal is sent to all processes in the process group associated with that typewriter. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs which read a typewriter and test for end-of-file on their input can terminate appropriately when hung up on.

Three other characters have a special meaning when typed. The ASCII DEL character (sometimes called ‘rubout’) is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control typewriter in the process group associated with that typewriter. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See *signal* (II).

The ASCII character FS generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated. If you find it hard to type this character, try control-\ or control-shift-L.

When not in raw mode, the ASCII SOH (control-A) character can be typed to temporarily stop output. Output is resumed when a second SOH or any other character is typed. SOH characters typed in this regard are not passed to the program.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals which respond to it from hanging up.

ioctl COMMANDS

ioctl commands 0 and 1 are valid for all line disciplines and can be used to fetch and set the line discipline.

Command 0: Fetch line discipline

```
ioctl(fildes, 000|('t'<<8), arg2)
struct {
  char unused, discipline;
} *arg2;
```

The current line discipline for the line specified by *fildes* is returned as *discipline.*

Command 1: Set line discipline; execute subcommand

```
ioctl(fildes, 001|('t'<<8), arg2)
struct {
  char subcmd, discipline;
  (remaining contents dependent
  on discipline and subcmd)
  .
} *arg2;
```

The line specified by *fildes* is set to operate according to *discipline* (if not already the case). Further action is taken as specified by *subcmd* and *discipline*. *subcmd=0* specifies no further action. Switching line discplines involves unsetting the old discipline and then setting the new one; procedures for unsetting and setting are programmed as part of the discipline.

Other *ioctl* commands are passed directly to a decoding routine for the line discipline currently in operation for actions specific to the line discipline. The following *ioctl* commands apply to line discipline *0.*

TYPEWRITER *ioctl* COMMANDS:

Command 1: Set line to discipline *0* (standard typewriter interface)

```
ioctl(fildes, 001|('t'<<8), arg2)
struct {
char      unused      0;
char      discp       0;
}*arg2;
```

The line specified by *fildes* is set to opeate according to discipline *0.* Modes and speeds are set to the default values obtained when a typewriter interface is initially opened. No subcommands have been defined.

Commands 10;11;12: *gtty(II)* and *stty(II)*

`ioctl(fildes, 010|('t'<<8), arg2)`	equivalent to *gtty(fildes, arg2)*
`ioctl(fildes, 011|('t'<<8), arg2)2`	equivalent to *stty(fildes, arg2)*
`ioctl(fildes, 012|('t'<<8), arg2)`	equivalent to *stty(fildes, arg2)* except that internal character buffers are not flushed prior to changing modes and/or speeds.

For compatibility the format for *arg2,* is the same for all of the above. See *stty(II)* and *gtty(II).*

Command 13: Fetch erase and kill characters

```
ioctl(fildes, 013|('t'<<8), arg2)
struct {
    int unused;
    char erase, kill;
} *arg2;
```

erase and *kill* are set to the erase and kill characters currently in use for the line.

Command 14: Set erase and kill characters

```
ioctl(fildes, 014|('t'<<8), arg2)
struct {
    int unused;
    char erase, kill;
} *arg2;
```

erase and *kill* replace the current erase and kill characters.

FILES

/dev/tty /dev/ln??

SEE ALSO

dc (IV), kl (IV), dh (IV), ioctl (II), getty (VIII), stty (I, II), gtty (II), signal (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a device of the proper type or the command is rejected by the line discipline. From C, a negative value indicates an error.

BUGS

On raw-mode output with standard typewriters, parity should be transmitted as specified in the characters written.

NAME

a.out — assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld.* Both programs make *a.out* executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, relocation bits and symbol table (in that order). The last two may be empty if the program was loaded with the ''−s'' option of *ld* or if the symbols and relocation have been removed by *strip.*

The header always contains 8 words:

1. A magic number (405, 407, 410, or 411(8))
2. The size of the program text segment
3. The size of the initialized portion of the data segment
4. The size of the uninitialized (bss) portion of the data segment
5. The size of the symbol table
6. The entry location
7. The relocation base address.
8. A flag indicating relocation bits have been suppressed

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at the relocation base (normally 0) in the core image; the header is not loaded. If the magic number (word 0) is 407, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. If the magic number is 411, the text segment is again pure, write-protected, and shared, and, moreover, instruction and data space are separated; the text segment begins at the relocation base and the data segment begins at location 0. See the 11/45 handbook for restrictions which apply to this situation. The magic number 405 indicates an overlay file. On execution, the current processes' data segment is replaced with the data segment from this module.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* system call.

The start of the text segment in the file is 20(8); the start of the data segment is $20+S_t$ (the size of the text) the start of the relocation information is $20+S_t+S_d$; the start of the symbol table is $20+2(S_t+S_d)$ if the relocation information is present, $20+S_t+S_d$ if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded. The next word is a flag indicating the type of symbol. The following values are possible:

00	undefined symbol
01	absolute symbol
02	text segment symbol
03	data segment symbol

04	bss segment symbol
37	file name symbol (produced by ld)
40	undefined external (.globl) symbol
41	absolute external symbol
42	text segment external symbol
43	data segment external symbol
44	bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

The last word of a symbol table entry contains the value of the symbol.

If the symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation bits for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the "suppress relocation" flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

00	indicates the reference is absolute
02	indicates the reference is to the text segment
04	indicates the reference is to initialized data
06	indicates the reference is to bss (uninitialized data)
10	indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on,* that the reference is relative to the pc (e.g., "clr x"); if *off,* that the reference is to the actual symbol (e.g., "clr *$x").

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO

as (I), ld (I), strip (I), nm (I), break (II)

ACCT(V) ACCT(V)

NAME

acct – Accounting file

DESCRIPTION

When a process terminates, an accounting record is written into one of the accounting files /usr/adm/acct, /usr/adm/sh_acct, or /usr/adm/su_acct if accounting has been activated. The file /usr/adm/acct is used if system accounting has been activated (see *accton* (VIII)), while /usr/adm/sh_acct or /usr/adm/su_acct is used if Shell accounting is being done. Any of these files may be summarized by using *sa* (VIII). All three files have the same format.

```
struct acct{
        char    ac_comm[14]   /* name of command */
        char    ac_flag       /* 0 if process execed, else 1 */
        char    ac_uid        /* real userid */
        long    ac_date       /* start time of command */
        long    ac_etime      /* elapsed time in seconds */
        long    ac_utime      /* user cpu time (1/60 sec) */
        long    ac_stime      /* system time (1/60 sec) */
};
```

SEE ALSO

sa (VIII), accton (VIII), acct (II)

NAME

ar – archive (library) file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld.*

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 177545(8) (chosen because it is unlikely to occur anywhere else). The header of each file is 26 bytes long:

0-13	file name, null padded on the right
14-17	modification time of the file
18	user ID of file owner
19	group ID of file owner
20-21	file mode
22-25	file size

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

SEE ALSO

ar (I), ld (I)

BUGS

CORE(V) CORE(V)

NAME

core − format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal (II)* for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called ‘‘core’’ and is written in the process’s working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system’s per-user data for the process, including the registers as they were at the time of the fault. The remainder represents the actual contents of the user’s core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

The format of the information in the first 1024 bytes is described by the *user* structure of the system. The important information not detailed there is the locations of the registers. Here are their offsets. The parenthesized numbers for the floating registers are used if the floating-point hardware is in single precision mode, as indicated in the status register.

fpsr	0004	
fr0	0006	(0006)
fr1	0036	(0022)
fr2	0046	(0026)
fr3	0056	(0032)
fr4	0016	(0012)
fr5	0026	(0016)
r0	1772	
r1	1766	
r2	1750	
r3	1752	
r4	1754	
r5	1756	
sp	1764	
pc	1774	
ps	1776	

In general, the debuggers *adb (I)* is sufficient to deal with core images.

SEE ALSO

adb (I), signal (II)

DIRECTORY (V) DIRECTORY (V)

NAME

dir – format of directories

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots.

By convention, the first two entries in each directory are for "." and "..". The first is an entry for the directory itself. The second is for the parent directory. The meaning of ".." is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases ".." has the same meaning as ".".

SEE ALSO

file system (V)

DUMP(V) DUMP(V)

NAME

dump − incremental dump tape format

DESCRIPTION

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of records which are an integral number of 512-byte blocks(usually 10 blocks per record). The first block of the first record has the following structure:

```
struct {
        int     isize;
        int     fsize;
        int     date[2];
        int     ddate[2];
        int     tsize;
        int     recsize;
};
```

Isize, and *fsize* are the corresponding values from the superblock of the dumped file system. (See *file system* (V).) *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date*. *Tsize* is the number of records per reel. *Recsize* is the number of 512-byte blocks per record. This block checksums to the octal value 031415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file exists, but was not dumped (was not modified after *ddate).* If the word is −1, the file does not exist. Other values for the word indicate that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see *file system* (V)) and also checksums to 031415. The two words next to the last word contain the i-number of the file and the tape record number, to aid in (unimplemented) recovery after tape errors. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a 'phase error'.

SEE ALSO

dump (VIII), restor (VIII), file system(V)

NAME

fs − format of file system volume

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 256-word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block*. Starting from its first word, the format of a super-block is

```
struct {
        int     isize;
        int     fsize;
        int     nfree;
        int     free[100];
        int     ninode;
        int     inode[100];
        char    flock;
        char    ilock;
        char    fmod;
        int     time[2];
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *Fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an ''impossible'' block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *free* array contains, in *free[1], ... , free[nfree−1],* up to 99 numbers of free blocks. *Free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *nfree,* and the new block is *free[nfree].* If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 100 words into the *free* array. To free a block, check if *nfree* is 100; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block's number and increment *nfree.*

Ninode is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode].* If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode.* If *ninode* is already 100, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

Flock and *ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node i is located in block $(i + 31) / 16$, and begins $32 \cdot ((i + 31) \pmod{16})$ bytes from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node is as follows:

```
struct {
        int     flags;            /* +0: see below */
        char    nlinks;           /* +2: number of links to file */
        char    uid;              /* +3: user ID of owner */
        char    gid;              /* +4: group ID of owner */
        char    size0;            /* +5: high byte of 24-bit size */
        int     size1;            /* +6: low word of 24-bit size */
        int     addr[8];          /* +8: block numbers or device number */
        int     actime[2];        /* +24: time of last access */
        int     modtime[2];       /* +28: time of last modification */
};
```

The flags are as follows:

```
 100000   i-node is allocated
 060000   2-bit file type:
     000000   plain file
     040000   directory
     020000   character-type special file
     060000   block-type special file.
 010000   large file
 004000   set user-ID on execution
 002000   set group-ID on execution
 000400   read (owner)
 000200   write (owner)
 000100   execute (owner)
 000070   read, write, execute (group)
 000007   read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files the high byte of the first address word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number n of a file is accessed as follows: n is divided by 512 to find its logical block number (say b) in the file. If the file is small (flag 010000 is 0), then b must be less than 8, and the physical block number is *addr[b]*.

If the file is large, b is divided by 256 to yield i. If i is less than 7, then *addr[i]* is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

For block b in a file to exist, it is not necessary that all blocks less than b exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck (VIII), check (VIII)

NAME

lines − script for the init process

DESCRIPTION

The *lines* file supplies the script to the *init* process (see *init* (VIII)). *init* runs at boot time, whenever one of its children terminates (e.g. when a user hangs up), and when requested by the superuser via *telinit* (VIII). Basically, it (re-)generates a process for each entry in the */dev/lines* file, which in most cases denotes a port to which a terminal may connect, but it may also identify processes not associated with a terminal line such as daemons.

The *lines* file consists of entries of the format:

id:runstate:action:shcommand:process

Comments in C syntax may precede or follow this format. The entry must not exceed 512 characters exclusive of comments. It is terminated by a new-line or carriage-return character, which may be escaped with a backslash (\).

Short entries of the format:

id:

are expanded to

id::**respawn::/etc/getty ln***id*

which is the most commonly desired action; it causes *getty* (VIII) to open file **/dev/ln***id.* This *getty* process is then respawned whenever the line process terminates. The entry fields are:

id	one or two characters except *xx, !B, RL, OT, NT* (see *init* (VIII)), used to uniquely identify the entry. By convention, for processes associated with terminal ports ("lines"), *id* should match the special file name **/dev/ln***id.* (See also above for the special shorthand notation *id:* for the entire line.) Also, it is good practice to use numerical *id* values for lines and nonnumerical values for daemons etc., which will aid in diagnosing line trouble.
runstate	must be either a subset of [0 1 2 3 4 5 6] or of [a b c] with no blanks or tabs in between, e.g. *:012:, :645:, :b:, :ac:,* or the empty set *::.* which is equivalent to *0123456.* When processing the entries of the *lines* file, *init* is in one of the system *runstates* 0 - 7. In single-user state, 7, all entries are ignored. In system *runstates* 0 - 6, *init* will process the entry only if one of the entry's *runstates* matches the system *runstate.* If the field contains a nonempty subset of [a b c], the entry is processed only if a *telinit* x command with matching x has been issued, independent of the system *runstate.*
action	must be one of the keywords: *respawn, once, wait, boot, bootwait, off, ondemand.* **respawn** will start or restart the process if it doesn't exist (i.e. at boot time or after it terminated), without waiting for the process to complete. **wait** and **once** will cause the entry to be processed once when the matching *runstate* is entered. Whereas **once** causes *init* to proceed to the next entry after having started the process, **wait** will make it wait for the process to terminate. Similarly, **boot** and **bootwait** will cause the entry to be processed only once at boot time (run states must match; *::* is the preferred coding). **bootwait** will cause *init* to await process completion, **boot** will let it proceed after starting the process. **off** will send the warning signal (*signal* SIGTRM) to the process if it is running, and wait 20 seconds before forcibly terminating the process via the kill signal (*signal* SIGKIL). **ondemand** should be coded with a, b, or c in the *runstate* field. The process is started

once in response to a *telinit* a, b, or c command.

shcommand	This optional field is treated as a shell command to be issued with the standard output directed to the console terminal (*/dev/ln00*). *Init* will wait for completion before advancing to the *process* field.
process	The named process is created by *init.* No files are normally opened for the process; however, the shell syntax for redirecting I/O *(<file, >file)* is recognized and causes the file descriptor(s) 0 and/or 1 to be assigned.

An entry is meaningless if both the *shcommand* and *process* fields are empty.

FILES

found in /etc

SEE ALSO

init(VIII), getty(VIII), exec(II), signal(II), open(II), telinit(VIII), sh(I)

PASSWD(V) PASSWD(V)

NAME

passwd − password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID (not used)
GCOS job number, box number, optional GCOS user-id
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

FILES

/etc/passwd

SEE ALSO

login (I), crypt (III), passwd (I)

SCCSFILE(V) SCCSFILE(V)

NAME

sccsfile − format of SCCS file

DESCRIPTION

Each record in an SCCS file is preceded by a one byte count, followed by that many bytes of data, followed by a pad byte if the count is even. The first byte of the file is also a pad byte. The count is stored offset from −128; i.e. a count of zero is stored as −128 and a count of 255 is stored as 127. The first byte of the file is null, but the other pad bytes contain garbage.

An SCCS file consists of four logical parts: the *header* contains assorted descriptive and control information about the file, the *release table* contains a count of the number of deltas in each release, the *delta table* contains information about each delta and the *body* contains the actual text records intermixed with control records. Each part is described below in terms of C structures.

Header. The header contains the following 226-byte structure:

```
struct Header {
        int  Hmagicno;
        char Htype[10];
        char Hpers[14];
        char Hdesc[100];
        int  Hfloor;
        int  Hceil;
        int  Hsw[5];
        int  Hrdef;
        char Hulist[32];
        char Hexpand[50];
        int  Hash;
};
```

The header fields have the following meanings. All ''char'' arrays except for *Hulist* must be null-terminated.

Hmagicno	This identifies SCCS files; it contains a binary 7.
Htype	This is used to document the contents of the file.
Hpers	This field is documentation only.
Hdesc	This field is documentation only.
Hfloor	The lowest release in which deltas may be added.
Hceil	The highest release in which deltas may be added. A zero is equivalent to 9999.
Hsw	Release locks. A non-zero value in one of the 5 elements locks the release whose number is that value.
Hrdef	The default release. A zero means that the default is determined from the release table in the manner described in *admin*
Hulist	A bit map on the user IDs 0 to 255; a 1 means the corresponding user can add deltas. All zeros is equivalent to all ones. To form the map, bytes are taken left-to-right within the 32-byte field and bits are taken right-to-left within each byte.

Hexpand	Room for expansion; must contain all zeros for now.
Hash	A hash count of the entire file. The hash count is determined by zeroing this field, appending a null byte to the file (SCCS files are always of odd length), and summing the words of the file using integer arithmetic. The result is then stored into this field.

Release table. Each release table record contains the following 4-byte structure:

```
struct Reltab {
                    int  Rrel;
                    int  Rlevs;
        };
```

There is one release table record for each release that has deltas, which gives the number of deltas in that release. If there are no deltas in a release that release need not appear in the release table at all, even if later releases do. The release records are kept in order, with the latest release first. The release table is terminated with an entry for release 0.

Delta table. Each delta table record contains the following structure. All "char" arrays must be null terminated.

```
struct Deltab {
                    int  Drel;
                    int  Dlev;
                    char Dtype;
                    char Dfill;;
                    int  Ddatetime[2];
                    char Dpgmr[8];
                    char Dhist[100];
        };
```

The length of a delta table record varies from 19 to 118 bytes, depending on the length of *Dhist.* The fields are as follows:

Drel	Release number.
Dlev	Level number.
Dtype	Type of delta: "D" if normal, "U" if unacknowledged non-propagating, "P" if acknowledged non-propagating, "R" if removed, "H" if changed history, and "C" if combined.
Dfill	Blank.
Ddatetime	The time, in the form returned by *time*
Dpgmr	The login name corresponding to the real user ID at the time the delta was created.
Dhist	History information associated with the delta.

Body. The body consists of text records and control records. Text records are stored without the newline character and are not null-terminated. Control records have the following structure:

```
struct Control {
                    int  Crel;
```

```
                int  Clev;
                char Cctl;
        };
```

Control records are 5 bytes long. There are three kinds of control records: *insert delete* and *end* represented by a *Cctl* of −11, −12 and −13 respectively. When text records are inserted, they are bracketed by insert and end controls for the release and level of the delta of which they are a part. Deleted text records are bracketed with delete and end controls. The body (and hence the SCCS file) is terminated by an end control for release 0, level 0.

SEE ALSO

prt(I), admin(I), get(I), delta(I),
The Source Code Control System TM75-9152-3. *SCCS/PWB User's Manual,* TM75-9144-1.

NAME

tp – DEC/mag tape formats

DESCRIPTION

The command *tp* dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See *boot procedures* (VIII).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

path name	32 bytes
mode	2 bytes
uid	1 byte
gid	1 byte
unused	1 byte
size	3 bytes
time modified	4 bytes
tape address	2 bytes
unused	16 bytes
check sum	2 bytes

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (file system (V)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 25 (resp. 63) on are available for file storage.

A fake entry (see *tp* (I)) has a size of zero.

SEE ALSO

file system (V), tp (I)

NAME

utmp – user information

DESCRIPTION

This file allows one to discover information about who is currently using UNIX. More specifically, it is an instantaneous record of current processes spawned by *init.* It provides the following three types of information:

(1) *init's* current run state
(2) *init's* last hardware reboot time
(3) *init's* direct descendants.

The file is recreated on every hardware boot and the information is kept in binary with each entry being 16 bytes long in the following format:

```
struct{
                char name[8];           /* user id or type 1 or type 2 data */
                int  line;              /* lid of each /etc/lines entry */
                int  time[2];           /* time of entry */
                int  pid;               /* process id */
};
```

FILES

/etc/utmp

SEE ALSO

init (VIII) and login (I), which maintain the file; who (I) which interprets it ; and lines (V), and wtmp (V).

NAME

wtmp – user login history

DESCRIPTION

This file maintains a running history on all of the processes spawned by *init*(UIII) and their subsequent deaths (including all terminal logins and logouts). Its format is exactly like *utmp*(V). Furthermore, this file will record two special entries whenever the system time-of-day is changed via the *date*(I) command. These two entries contain the time when the date was reset and the new system time (see *who*(I)).

Wtmp is maintained by *login* (I) and *init* (VIII). Neither of these programs creates the file, so if it is removed, record-keeping is turned off. It is summarized by *ac* (VIII) and interpreted by *who(I)*.

FILES

/etc/wtmp

SEE ALSO

utmp (V), login (I), init (VIII), ac (VIII), who (I), date(I)

NAME

agen – generate associative memory drivers

SYNOPSIS

agen file

DESCRIPTION

Agen permits easy referencing of table lookups, etc.

The input consists of a series of table definitions followed by %% followed by programs. Each definition is of the form

name (accessing) object ;

where name is the name of a table to be searched; accessing is a description of the subscripts used, being a list of the choices 'string', 'int' and 'char'. Object is an example of the things in the table. It is used only to get the size in bytes; it may be replaced by the number corresponding to sizeof(object). Alternatively one of the accessing arguments may be 'size' indicating that the number of bytes required will be given as that argument. The 'exist' argument may be given; if it is, the corresponding argument of the search function is 0 for normal operation and 1 if failure to find the item should return 0 rather than a pointer to new storage for it. An 'exist' argument of -1 implies that the corresponding object should be deleted. Also, an argument of 'size' may be given; this indicates that the size of the object to be stored will be supplied in the call (in bytes).

It is also possible to specify a table size and a method.

name [length] ["hash" | "binary"] (accessing) object ;

specifies either a hash table or a binary tree search. For a hash search the length must be given.

BUGS

The lookup routines are slow.

NAME

bj – the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

> A player 'natural' (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a 'push' (no money exchange).
>
> If the dealer has an ace up, the player is allowed to make an 'insurance' bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.
>
> If the player is dealt two cards of the same value, he is allowed to 'double'. He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)
>
> If a dealt hand has a total of ten or eleven, the player may 'double down'. He may double the bet ($2 to $4) and receive exactly one more card on that hand.
>
> Under normal play, the player may 'hit' (draw a card) as long as his total is not over twenty-one. If the player 'busts' (goes over twenty-one), the dealer wins the bet.
>
> When the player 'stands' (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.
>
> If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new line for 'yes', or just new line for 'no'.

?   (means, "do you want a hit?")
Insurance?
Double down?

Every time the deck is shuffled, the dealer so states and the 'action' (total bet) and 'standing' (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

BUGS

NAME

cal — print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.

CHESS (VI) CHESS (VI)

NAME

chess – the game of chess

SYNOPSIS

/usr/games/chess

DESCRIPTION

Chess is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

FILES

/usr/lib/book	opening 'book'

DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

WARNING

Over-use of this program will cause it to go away.

BUGS

Pawns may be promoted only to queens.

CUBIC (VI) CUBIC (VI)

NAME

cubic − three dimensional tic-tac-toe

SYNOPSIS

/usr/games/cubic

DESCRIPTION

Cubic plays the game of three dimensional 4×4×4 tic-tac-toe. Moves are given by the three digits (each 1-4) specifying the coordinate of the square to be played.

WARNING

Too much playing of the game will cause it to disappear.

BUGS

FACTOR (VI) FACTOR (VI)

NAME

factor − discover prime factors of a number

SYNOPSIS

factor

DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If *factor* is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime. It takes 1 minute to factor a prime near 10^{13}.

DIAGNOSTICS

'Ouch.' for input out of range or for garbage input.

BUGS

FED (VI) FED (VI)

NAME

fed − edit form letter memory

SYNOPSIS

fed

DESCRIPTION

Fed is used to edit a form letter associative memory file, **form.m**, which consists of named strings. Commands consist of single letters followed by a list of string names separated by a single space and ending with a new line. The conventions of the Shell with respect to '*' and '?' hold for all commands but **m**. The commands are:

e name ...
: writes the string whose name is *name* onto a temporary file and executes *ed.* On exit from the *ed* the temporary file is copied back into the associative memory. Each argument is operated on separately. Be sure to give an editor *w* command (without a filename) to rewrite *fed's* temporary file before quitting out of *ed.*

d [name ...]
: deletes a string and its name from the memory. When called with no arguments **d** operates in a verbose mode typing each string name and deleting only if a **y** is typed. A **q** response returns to *fed*'s command level. Any other response does nothing.

m name1 name2 ...
: (move) changes the name of name1 to name2 and removes previous string name2 if one exists. Several pairs of arguments may be given. Literal strings are expected for the names.

n [name ...]
: (names) lists the string names in the memory. If called with the optional arguments, it just lists those requested.

p name ...
: prints the contents of the strings with names given by the arguments.

q
: returns to the system.

c [**p**] [**f**]
: checks the associative memory file for consistency and reports the number of free headers and blocks. The optional arguments do the following:

p causes any unaccounted-for string to be printed.

f fixes broken memories by adding unaccounted-for headers to free storage and removing references to released headers from associative memory.

FILES

/tmp/ftmp?	temporary
form.m	associative memory

SEE ALSO

form (VI), ed (I), sh (I)

WARNING

It is legal but unwise to have string names with blanks, '*' or '?' in them.

BUGS

NAME

form – form letter generator

SYNOPSIS

form proto arg ...

DESCRIPTION

Form generates a form letter from a prototype letter, an associative memory, arguments and in a special case, the current date.

If *form* is invoked with the *proto* argument *x*, the associative memory is searched for an entry with name *x* and the contents filed under that name are used as the prototype. If the search fails, the message '[*x*]:' is typed on the console and whatever text is typed in from the console, terminated by two new lines, is used as the prototype. If the prototype argument is missing, '{letter}' is assumed.

Basically, *form* is a copy process from the prototype to the output file. If an element of the form [*n*] (where *n* is a digit from 1 to 9) is encountered, the *n*-th *arg* is inserted in its place, and that argument is then rescanned. If [0] is encountered, the current date is inserted. If the desired argument has not been given, a message of the form '[*n*]:' is typed. The response typed in then is used for that argument.

If an element of the form [*name*] or {*name*} is encountered, the *name* is looked up in the associative memory. If it is found, the contents of the memory under this *name* replaces the original element (again rescanned). If the *name* is not found, a message of the form '[*name*]:' is typed. The response typed in is used for that element. The response is entered in the memory under the name if the name is enclosed in [ ]. The response is not entered in the memory but is remembered for the duration of the letter if the name is enclosed in {}. Brackets and braces may be nested.

In both of the above cases, the response is typed in by entering arbitrary text terminated by two new lines. Only the first of the two new lines is passed with the text.

If one of the special characters [{]}\ is preceded by a \, it loses its special character.

If a file named 'forma' already exists in the user's directory, 'formb' is used as the output file and so forth to 'formz'.

The file 'form.m' is created if none exists. Because form.m is operated on by the disc allocator, it should only be changed by using *fed,* the form letter editor, or *form*.

FILES

form.m	associative memory
form?	output file (read only)

SEE ALSO

fed (VI), nroff (I)

BUGS

An unbalanced] or } acts as an end of file but may add a few strange entries to the associative memory.

NAME

gsi − interpret extended character set on GSI terminal

SYNOPSIS

gsi

DESCRIPTION

Gsi interprets special characters understood by the Model 37 Teletype terminal and turns them into the escape sequences understood by the GSI and other Diablo-based terminals. The things interpreted include vertical motions and extended graphic characters. It is most often used in a pipeline like

```
nroff file ... | gsi
```

SEE ALSO

BUGS

Some funny characters can't be correctly printed in column 1 because you can't move to the left from there.

NAME

hyphen − find hyphenated words

SYNOPSIS

hyphen file ...

DESCRIPTION

It finds all of the words in a document which are hyphenated across lines and prints them back at you in a convenient format.

If no arguments are given, the standard input is used. Thus *hyphen* may be used as a filter.

BUGS

Yes, it gets confused, but with no ill effects other than spurious extra output.

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [-code] file

DESCRIPTION

Lex generates programs for simple lexical analysis.

The input file contains strings and expressions to be searched for, and C text to be executed when found. A file lex.yy.c is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The strings may contain square brackets to indicate character classes, as in

```
[abx-z]
```

to indicate a, b, x, y, and z; and the operators *, +, and ? mean respectively any non-negative number of, any positive number of, and either zero or one occurences of, the previous character or character class. The character '.' is the class of all ascii characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character $ at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in yylextext, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus

```
[a-zA-Z]+
```

matches a string of letters. The actual string matched is left in yylextext, an external character array. Matching is done in order of the strings in the file.

Three subroutines are expected: input() to read a character; unput(c) to replace a character read; and output(c) to place an output character. The library defines these in terms of the standard streams (and the -lp portable library), but you can replace them. The program generated is named yylex(), and the library contains a main() which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function yymore() accumulates additional characters into the same yylextext; and the function yyless(p) pushes back the portion of the string matched beginning at p, which should be between yylextext and yylastch.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the lex.yy.c file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a non-blank define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done. Example:

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yylextext);
0{D}+   printf("octal number %s\n",yylextext);
{D}+    printf("decimal number %s\n",yylextext);
"++"            printf("unary op\n");
"+"             printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                default: go to loop;
                }
        }
```

The external names generated by *lex* all begin with the "code" string from the optional first argument; if omitted it is "yy". Furthermore,the name of the output file is "lex.code.c" in the general case. The code argument may be given on the first line of the input file as %-code.

FILES

/lib/libl.a (-ll) library

BUGS

A right context portion of an expression (past a /) may not contain the operators *, +, ? | or (.

NAME

moo − guessing game

SYNOPSIS

/usr/games/moo

DESCRIPTION

Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A 'cow' is a correct digit in an incorrect position. A 'bull' is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

BUGS

NAME

ptx – permuted index

SYNOPSIS

ptx [**−t**] input [output]

DESCRIPTION

Ptx generates a permuted index from file *input* on file *output.* It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally the sorted lines are rotated so the keyword comes at the middle of the page.

Input should be edited to remove useless lines. The following words are suppressed: 'a', 'an', 'and', 'as', 'is', 'for', 'of', 'on', 'or', 'the', 'to', 'up', 'all', 'at', 'data', 'do', 'in', 'into', 'when'.

The optional argument **−t** causes *ptx* to prepare its output for the phototypesetter.

The index for this manual was generated using *ptx*.

FILES

/bin/sort

NAME

reform − reformat text file

SYNOPSIS

reform [*tabspec1* [*tabspec2*]] [**+b***n*] [**+e***n*] [**+f**] [**+i***n*] [**+m***n*] [**+p***n*] [**+s**] [**+t***n*]

DESCRIPTION

Reform reads each line of the standard input file, reformats it, and then writes it to the standard output. Various combinations of reformatting operations can be selected, of which the most common involve rearrangement of tab characters.

Reform first scans any arguments, which may be given in any order. It then processes its input file, performing the following actions upon each line, in the order given.

- A line is read from the standard input.
- If **+s** is given, the first 10 characters of the line are stripped off. Characters 1−4 (SCCS Release) and 6−9 (SCCS Level) are saved for later addition to the end of the line.
- The line is expanded into a tabless form, by replacing tabs with blanks according to the input tab specification *tabspec1*.
- If **+p***n* is given, *n* blanks are prepended to the line.
- If **+t***n* is given, the line is truncated to a length of *n* characters.
- All trailing blanks are now removed.
- If **+e***n* is included, the line is extended out with blanks to the length of *n* characters.
- If **+s** is given, the previously saved SCCS Release and Level are added to the end of the line.
- If **+b***n* is given, the *n* characters at the beginning of the line are converted to blanks, if and only if all of them are either digits or blanks.
- If **+m***n* is included, the line is moved left, i.e., *n* characters are removed from the beginning of the line.
- The line is now contracted by replacing some blanks with tab characters according to the list of tabs indicated by the output tab specification *tabspec2*, and is written to the standard output file. Option **+i** controls the method of contraction (see below).

The various arguments accepted by *reform* are as follows:

tabspec1	describes the tab stops assumed for the input file. This tab specification may take on any of the forms described below. In addition, the operand −− indicates that the tab specification is to be found in the first line read from the standard input. If no legal tab specification is found there, −8 is assumed. If *tabspec1* is omitted entirely, −− is assumed.
tabspec2	describes the tabs assumed for the output file. It is interpreted the same as *tabspec1*, except that omission of *tabspec2* causes the value of *tabspec1* to be used for *tabspec2*.

The remaining arguments are all optional and may be used in any combination, although only a few combinations make much sense. Specifying an argument causes an action to be performed, as opposed to a usual default of not performing the action. Some options include numeric values, which also have default values. Option actions are applied to each line in the order described above. Any line length mentioned applies to the length of a line just previous to the execution of the option described, and the terminating newline is never counted in the line length.

+b*n* causes the first *n* characters of a line to be converted to blanks, if and only if those characters include only blanks and digits. If *n* is omitted, the default value is 6, which is useful in deleting sequence numbers from COBOL programs.

+e*n* causes each line shorter than *n* characters to be extended out with blanks to that length. Omitting *n* implies a default value of 72. This option is useful for those rare cases in which some sequence numbers need to be added to an existing unnumbered file, e.g., the use of $ in editor regular expressions is more convenient if all lines have equal length. I.e., the user intends to issue editor commands like:

s/$/00001000/

+f causes a format line to be written to the standard output, preceding any other lines written. The format line is taken from *tabspec2*, i.e., the line normally appears as follows:

<:t−*tabspec2* **d:>**

+i*n* controls the technique used to compress interior blanks into tabs. Unless this option is specified, any sequence of 1 or more blanks may be converted to a single tab character if it occurs in an appropriate place. This causes no problems for blanks occurring before the first nonblank character in a line, and it is always possible to convert the result back to an equivalent tabless form. However, occasionally an interior blank (one occurring after the first nonblank) is converted to a tab when this is not intended. For instance, this might occur in any program written in a language utilizing blanks as delimiters. Any single blank might be converted to a tab if it occurs just before a tab stop. Insertion or deletion of characters preceding such a tab may cause it to be interpreted in an unexpected way at a later time. If the **+i** option is used, no string of blanks may be converted to a tab unless there are *n* or more contiguous blanks. The default value is 2. Note that leading blanks are always converted to tabs when possible.

+m*n* causes each line to be moved left *n* characters, with a default value of 6. This can be useful for crunching COBOL programs.

+p*n* causes *n* blanks to be prepended (default of 6 if omitted). This option is effectively the inverse of **+m***n*, and is often useful for adjusting the position of *nroff(I)* output for terminals lacking both forms tractor positioning and settable left margin.

+s is used with the **−m** option of *get(I)*. The **−m** option causes *get* to prepend SCCS Release and Level numbers to each generated line. The **+s** option causes these numbers to be removed from the front of each line, saved, and then later added to the end of the line. Because **+e72** is implied by this option, the effect is to produce 80-character card images with SCCS Release and Level in columns 73−80.

+t*n* causes any line longer than *n* characters to be truncated to that length. If *n* is omitted, the length defaults to 72. Sequence numbers can thus be removed, also permitting additional preceding blanks to be deleted.

Three types of tab specification are accepted for *tabspec:* "canned", repetitive, and arbitrary.

−code	gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:
−a	1,10,16,36,72 Assembler, IBM S/370, first format
−a2	1,10,16,40,72 Assembler, IBM S/370, second format
−c	1,8,12,16,20,55 COBOL, normal format
−c2	1,6,10,14,49 COBOL compact format (columns 1−6 omitted).
−cx	1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67 COBOL compact format (columns 1−6 omitted), with more tabs than **−c2**.
−f	1,7,11,15,19,23 FORTRAN
−p	1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61 PL/I
−s	1,10,55 SNOBOL
−u	1,12,20,44 UNIVAC 1100 Assembler

In addition to these "canned" formats, three other types exist:

−n	a repetitive specification requests tabs at columns $1+n$, $1+2*n$, etc.
n1,n2,...	the arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 20 numbers are allowed. The maximum tab value accepted is 158. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered identical.

DIAGNOSTICS

All diagnostics are fatal, and the offending line is displayed following the message.
"line too long" if any line exceeds 512 characters (in tabless form).
"not SCCS -m" if first 10 characters of line are not in proper format when **+s** flag used.

EXIT CODES

0 − normal
1 − any error

SEE ALSO

get(I), nroff(I)

BUGS

Reform is aware of the meanings of Backspace and Escape sequences, so that it can be used as a postprocessor for *nroff*. However, be warned that the **+e, +m, +t** options only count characters, not positions. Anyone using these options on output containing backspaces or halfline motions will probably obtain unexpected and useless results.

NAME

sno − Snobol interpreter

SYNOPSIS

sno [file]

DESCRIPTION

Sno is a Snobol III (with slight differences) compiler and interpreter. *Sno* obtains input from the concatenation of *file* and the standard input. All input through a statement containing the label 'end' is considered program and is compiled. The rest is available to 'syspit'.

Sno differs from Snobol III in the following ways.

There are no unanchored searches. To get the same effect:

a ** b	unanchored search for b
a *x* b = x c	unanchored assignment

There is no back referencing.

x = "abc"	
a *x* x	is an unanchored search for 'abc'

Function declaration is different. The function declaration is done at compile time by the use of the label 'define'. Thus there is no ability to define functions at run time and the use of the name 'define' is preempted. There is also no provision for automatic variables other than the parameters. For example:

define f( )

or

define f(a,b,c)

All labels except 'define' (even 'end') must have a non-empty statement.

If 'start' is a label in the program, program execution will start there. If not, execution begins with the first executable statement. 'define' is not an executable statement.

There are no built-in functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators '/' and '*' must be set off by space.

The right side of assignments must be non-empty.

Either ′ or " may be used for literal quotes.

The pseudo-variable 'sysppt' is not available.

SEE ALSO

Snobol III manual. (JACM; Vol. 11 No. 1; Jan 1964; pp 21)

BUGS

NAME

tbl − format tables for nroff or troff

SYNOPSIS

tbl [files] ...

DESCRIPTION

Tbl is an nroff (I) or troff(I) preprocessor for formatting tables. The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and reformatted. There are several global options; if any are desired they are specified on the first line after .TS as a series of keywords separated by blanks or commas and followed by a semicolon. The possible words are:

center	center the table
expand	format the table to fill the current line length
box	enclose te table in a box
allbox	draw all possible lines so that every item is in a box

After this line (or after .TS if no global options are given) are the lines describing the table format. Each line describes a line of the actual table. One letter is used for each column. As many lines as are needed to describe the table are given; the last line should end with the character '.' to signal the end of the format information. The last line of the description will apply to all following lines of the table. The legal characters to describe a column are:

c	center within the column
r	right-adjust
l	left-adjust
n	numerical adjustment: the units digits of numbers are aligned.
s	span the previous entry over this column.
_	replace this entry with a horizontal line.
=	replace this entry with a double horizontal line.

A column letter may be followed by an integer giving the number of spaces between this column and the next; 3 is default. A column letter may be preceded by a '|' character to indicate that a vertical line is to be drawn to the left of this column. Letting \t represent a tab (which must be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

Household Population		
Town	Households	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Branchburg	1644	3.49
Bridgewater	7897	3.81
Far Hills	240	3.19

If a table element contains only "_" or "=" a single or double line (respectively) is drawn across the column at that point. If a table line contains only "_" or "=", a line is drawn all the way across the table (resp. double line). If a column describer contains the character "|", a vertical line is drawn to the left of that column beginning at the point in the column corresponding to the position of the vertical bar in the describer and extending to the bottom of the table.

If no arguments are given, *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn* or *neqn,* the *tbl* command should be first to minimize the volume of data passed through pipes.

SEE ALSO

"TBL – A Program to Format Tables", by M.E.Lesk (outside BTL: "Computer Science Technical Report #49", available from BTL, Computer Information Services, Murray Hill, N.J.)

NAME

tmac − macros for formatting manuscripts

SYNOPSIS

nroff −ms [options] file ...

DESCRIPTION

This package of *nroff* macro definitions provides a canned formatting facility for technical papers.

The package supports three different formats: BTL technical memorandum with cover sheet, released paper with cover sheet, and an abbreviated 'debugging' form without cover sheet.

The macro requests are defined in the attached Request Reference. Many *nroff* requests are unsafe in conjunction with this package, however the requests listed below may be used with impunity after the first .PP.

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin

Output of the *tbl* (I) preprocessor for tables is acceptable as input.

FILES

/usr/lib/tmac.s

SEE ALSO

nroff (I)

BUGS

REQUEST REFERENCE

Request Form	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AU *x y*	no	yes	Author's name follows. *x* is location and *y* is extension, ignored except in TM.
.B	no	no	Boldface text follows.
.CS *x*...	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.DA *x*	nroff	no	'Date line' at bottom of page is *x*. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered. Implies .KS.
.EN	-	yes	Space after equation produced by *eqn* or *neqn*.
.EQ *x*	-	yes	Space before equation. Equation number is *x*.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HO	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I	no	no	Italic text follows.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.NH *n*	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's measured from current indentation.
.SG *x*	no	yes	Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TL	no	yes	Title follows.
.TM *x y z*	no	-	BTL TM cover sheet and first page, *x*=TM number, *y*=(quoted list of) case number(s), *z*=file number. Must precede other requests.
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

NAME

ttt − the game of tic-tac-toe

SYNOPSIS

/usr/games/ttt

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to know the game completely.

FILES

/usr/games/ttt.k	learning file

BUGS

NAME

wump — the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of ‘‘Hunt the Wumpus.’’ A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company, 2,* 2 (November 1973).

BUGS

It will never replace Space War.

ASCII (VII) ASCII (VII)

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 "	043 #	044 $	045 %	046 &	047 ´
050 (	051)	052 *	053 +	054 ,	055 –	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 del

FILES

found in /usr/pub

GREEK (VII) GREEK (VII)

NAME

greek − graphics for extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek

DESCRIPTION

Greek gives the mapping from ascii to the "shift out" graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. It contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	φ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	¬	_
partial	∂	]	integral	∫	^			

SEE ALSO

ascii (VII)

NAME

mtab – mounted file system table

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last "/" is thrown away.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount (VIII), umount (VIII)

BUGS

NAME

tabs – set tab stops

SYNOPSIS

cat /usr/pub/tabs

DESCRIPTION

Printing this file on a suitable terminal sets tab stops every 8 columns. Suitable terminals include the Teletype model 37 and the GE TermiNet 300.

These tab stop settings are desirable because UNIX assumes them in calculating delays.

NAME

ac — login accounting

SYNOPSIS

ac [**−w** wtmp] [**−p**] [**−d**] people

DESCRIPTION

Ac produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. **−w** is used to specify an alternate *wtmp* file. **−p** prints individual totals; without this option, only totals are printed. **−d** causes a printout for the last midnight to midnight period. Any *people* after **−p** will limit the printout to only the specified login names. If no *wtmp* file is given, */etc/wtmp* is used.

The accounting file */etc/wtmp* is maintained by *init* and *login*. Neither of these programs creates the file, so if it does not exist, no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand, if the file is left undisturbed, it will grow without bound, so periodically any information desired should be collected and the file truncated.

FILES

/etc/wtmp

SEE ALSO

init (VIII), login (I), wtmp (V).

BUGS

ACCTON(VIII) ACCTON(VIII)

NAME

accton — turn accounting on/off

SYNOPSIS

/etc/accton [file]

DESCRIPTION

System accounting information is placed at the end of *file.* If no *file* argument is given or if the file does not exist, system accounting is turned off.

SEE ALSO

acct (II), sa (VIII)

BUGS

BOOT PROCEDURES(VIII) BOOT PROCEDURES(VIII)

NAME

boot procedures − UNIX startup

DESCRIPTION

The advent of the 11/70 and its associated peripherals has changed the boot procedures. The following procedures apply only to C-language systems.

How to start UNIX. UNIX is started by placing it in core starting at location zero and transferring to zero. There are various ways to do this.

The *tp*(I) command places a bootstrap program on the otherwise unused block zero of the tape. The DECtape version of this boot program is called *tboot;* the magnetic tape version is called *mboot.* If *tboot* or *mboot* is read into location zero and executed there, it will type '=' on the console, read a *tp*(I) entry name, load that entry into core, and transfer to zero. Thus one way to start UNIX is to maintain the UNIX code on a tape using *tp*(I). (Caution: the file /usr/mdec/tboot (DECtape) or /usr/mdec/mboot (magtape) must be present when the tape is made or updated.) Booting is then accomplished by executing a program which reads in and jumps to the first block of the tape. The standard DEC ROM which loads DECtape is sufficient to read in *tboot,* but the magtape ROM loads block one, not zero. If no suitable ROM is available, magtape and DECtape programs are presented below which may be manually placed in core and executed. In response to the '=' prompt, type the entry name of the system on the tape (`unix' is suggested) It is strongly recommended that a current version of the system be maintained in this way, even if another method of booting the system is usually used.

Another method of booting the system involves the otherwise unused block zero of each UNIX file system. One of four separate boot programs can be used in this method. The single-block program *uboot* reads a single character (either **p** or **k** for RP03 or RK04/05, drive 0) to specify which device is to be searched. The other boot programs are called *rkboot, rpboot,* and *hpboot.* These programs are also one block long and they also read one character from the console, but they are designed to search one device type (drive zero in all cases), not one of two. *Rkboot* accepts the character **k** and searches the RK04/05 disk. *Rpboot* accepts the character **p** and searches the RP03 disk. *Hpboot* accepts the character **4** and searches the RP04 disk. All four bootstrap programs will then read a UNIX pathname from the console, find the corresponding file on the device, load that file into core location zero, and transfer to it. *Uboot, rkboot, rpboot,* and *hpboot* operate under very severe space constraints. They supply no prompts, except a carriage return and line feed that are echoed after the **p**, **k**, or **4**. No diagnostic is provided if the indicated file cannot be found, nor is there any means of correcting typographical errors in the file name except to start the program over. These four bootstrap programs can reside in block zero of the device(s) they are capable of searching, or they may be loaded from a *tp* tape as described above. The correct bootstrap program can be placed on block zero at system generation by the *mkfs* (VIII) command or it can be placed there any time after that with the *cp* (I) command.

The standard DEC disk ROMs will load and execute *uboot, rkboot, rpboot,* and *hpboot* from block zero of the device.

The switches. The console switches play an important role in the use and especially the booting of UNIX. During operation, the console switches are examined 60 times per second, and the contents of the address specified by the switches are displayed in the data display register only if the data display select knob is set to DISPLAY REGISTER. (This is not true on the 11/40 since there is no display register on that machine.) If the switch address is even, the address is interpreted in kernel (system) space; if odd, the rounded-down address is interpreted in the current user space.

If any diagnostics are produced by the system, they are printed on the console only if the switches are non-zero. Thus it is wise to have a non-zero value in the switches at all times.

During the startup of the system, the *init* program (VIII) reads the switches and will come up single-user if the switches are set to 000007 (0000007 on an 11/70).

It is unwise to have a non-existent address in the switches. This causes a bus error in the system (displayed as 177777) at the rate of 60 times per second. If there is a transfer of more than 16ms duration on a device with a data rate faster than the bus error timeout (approx 10μs), then a permanent disk non-existent-memory error will occur.

ROM programs. Below are given some programs which are suitable for installing in read-only memories, or for manual keying into core if no ROM is present. Each program is position-independent but should be placed well above location 0 so it will not be overwritten. Each reads a block from the beginning of a physical device into core location zero. The octal words constituting the program are listed on the left.

DECtape (drive 0) without search:

```
012700          mov     $tcba,r0
177346
010040          mov     r0,-(r0)                / use tc addr for wc
012710          mov     $3,(r0)                 / read bn forward
000003
105710    1:    tstb    (r0)                    / wait for ready
002376          bge     1b
112710          movb    $5,(r0)                 / read (forward)
000005
000777          br                              / loop; now halt and start at 0
```

DECtape (drive 0) with search:

```
012700    1:    mov     $tcba,r0
177346
010040          mov     r0,-(r0)                / use tc addr for wc
012740          mov     $4003,-(r0)             / read bn reverse
004003
005710    2:    tst     (r0)
002376          bge     2b                      / wait for error
005760          tst     -2(r0)                  / loop if not end zone
177776
002365          bge     1b
012710          mov     $3,(r0)                 / read bn forward
000003
105710    2:    tstb    (r0)                    / wait for ready
002376          bge     2b
112710          movb    $5,(r0)                 / read (forward)
000005
105710    2:    tstb    (r0)                    / wait for ready
002376          bge     2b
005007          clr     pc                      / transfer to zero
```

Caution: both of these DECtape programs will (literally) blow a fuse if 2 drives are dialed to zero.

Magtape (TU10) from load point:

```
012700          mov     $mtcma,r0
172526
010040          mov     r0,-(r0)                / usr mt addr for wc
012740          mov     $60003,-(r0)            / read 9-track
060003
000777          br                              / loop; now halt and start at 0
```

Magtape (TU16) from load point:

The numbers in parentheses are for an 11/70

Zero the following addresses

772442 (1772442)
772444 (1772444)
772446 (1772446)

Halt and load address 772472 (1772472)
Set switches to 001300 (0001300)
Load address 772440 (1772440)
Set switches to 000071 (0000071)
Enable and deposit

RK (drive 0):

```
012700          mov     $rkmr,r0
177414
005040          clr     -(r0)
005040          clr     -(r0)
010040          mov     r0,-(r0)
012740          mov     $5,-(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

RP03 (drive 0):

```
012700          mov     $rpmr,r0
176726
005040          clr     -(r0)
005040          clr     -(r0)
005040          clr     -(r0)
010040          mov     r0,-(r0)
012740          mov     $5,-(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

RP04 (drive 0):

For an 11/70:

Halt and load address 1765000
Set switches to 0000070
Enable and start

For an 11/45 with optional ROM:

Halt and load address 773350
Enable and start

FILES

/unix – UNIX code
/usr/mdec/mboot – *tp* magtape bootstrap
/usr/mdec/tboot – *tp* DECtape bootstrap
/usr/mdec/uboot – file system bootstrap
/usr/mdec/tu/rkboot – RK04/05 file system bootstrap
/usr/mdec/tu/rpboot – RP03 file system bootstrap
/usr/mdec/tu/hpboot – RP04 file system bootstrap

SEE ALSO

tp (I), init (VII), mkfs (VIII)

NAME

check − file system consistency check

SYNOPSIS

check [**−lsuib** [numbers]] [filesystem]

DESCRIPTION

Check examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. It also compares the link-count for each allocated i-number (i.e., i-node) with the number of references (i.e., directory entries) to it. If the file system is not specified, a check of a default file system is performed. The normal output of *check* includes a report of

the number of blocks missing (i.e., not
in any file nor in the free list),
the number of special files,
the total number of files,
the number of large files,
the number of directories,
the number of indirect blocks,
the number of blocks used in files,
the highest-numbered block appearing in a file,
the number of free blocks.

The −l flag causes *check* to produce as part of its output report a list of all the path names of files on the file system. The list is in i-number order; the first name for each file gives the i-number while subsequent names (i.e., links) have the i-number suppressed. The entries "." and ".." for directories are also suppressed. If the flag is given as **−ll**, the listing will include the accessed and modified times for each file. The −l option supersedes **−s.**

The −s flag causes *check* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example, if the root file system has to be salvaged), care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. However, a *sync* (VIII) prior to the reboot will undo the salvage. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled, these words will have to be patched. The −s flag causes the normal output reports to be suppressed.

The occurrence of **i** *n* times in a flag argument, **−ii...i,** causes *check* to store away the next *n* arguments which are taken to be i-numbers. When any of these i-numbers is encountered in a directory, a diagnostic is produced, as described below, which indicates among other things the entry name.

Likewise, *n* appearances of **b** in a flag, like **−bb...b,** cause the next *n* arguments to be taken as block numbers which are remembered; whenever any of the named blocks turns up in a file, a diagnostic is produced.

FILES

Currently, /dev/rk0 is the default file system.

SEE ALSO

file system (V), clri (VIII), restor (VIII)

DIAGNOSTICS

If a read error is encountered, the block number of the bad block is printed and *check* exits. ''Bad freeblock'' means that a block number outside the available space was encountered in the free list. ''*n* dups in free'' means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

An important class of diagnostics is produced by a routine which is called for each block which is encountered in an i-node corresponding to an ordinary file or directory. These have the form

b# *complaint* **; i=** *i#* (*class*)

Here *b#* is the block number being considered; *complaint* is the diagnostic itself. It may be

blk	if the block number was mentioned as an argument after **−b;**
bad	if the block number has a value not inside the allocatable space on the device, as indicated by the devices's super-block;
dup	if the block number has already been seen in a file;
din	if the block is a member of a directory, and if an entry is found therein whose i-number is outside the range of the i-list on the device, as indicated by the i-list size specified by the super-block. Unfortunately this diagnostic does not indicate the offending entry name, but since the i-number of the directory itself is given (see below) the problem can be tracked down.

The *i#* in the form above is the i-number in which the named block was found. The *class* is an indicator of what type of block was involved in the difficulty:

sdir	indicates that the block is a data block in a small file;
ldir	indicates that the block is a data block in a large file (the indirect block number is not available);
idir	indicates that the block is an indirect block (pointing to data blocks) in a large file;
free	indicates that the block was mentioned after **−b** and is free;
urk	indicates a malfunction in *check.*

When an i-number specified after **−i** is encountered while reading a directory, a report in the form

i# **ino; i=** *d#* (*class*) *name*

where *i#* is the requested i-number, *d#* is the i-number of the directory, *class* is the class of the directory block as discussed above (virtually always ''sdir'') and *name* is the entry name. This diagnostic gives enough information to find a full path name for an i-number without using the **-l** option: use **−b** *n* to find an entry name and the i-number of the directory containing the reference to *n,* then recursively use **−b** on the i-number of the directory to find its name.

Another important class of file system diseases indicated by *check* is files for which the number of directory entries does not agree with the link-count field of the i-node.

The diagnostic is hard to interpret. It has the form

i# delta

Here *i#* is the i-number affected. *Delta* is an octal number accumulated in a byte, and thus can have the value 0 through 377(8). The easiest way (short of rewriting the routine) of explaining the significance of *delta* is to describe how it is computed.

If the associated i-node is allocated (that is, has the *allocated* bit on), add 100 to *delta*. If its link-count is non-zero, add another 100 plus the link-count. Each time a directory entry specifying the associated i-number is encountered, subtract 1 from *delta*. At the end, the i-number and *delta* are printed if *delta* is neither 0 nor 200. The first case indicates that the i-node was unallocated and no entries for it appear; the second that it was allocated and that the link-count and the number of directory entries agree.

Therefore (to explain the symptoms of the most common difficulties) *delta* = 377 (−1 in 8-bit, 2's complement octal) means that there is a directory entry for an unallocated i-node. This is somewhat serious and the entry should be be found and removed forthwith. *Delta* = 201 usually means that a normal, allocated i-node has no directory entry. This difficulty is much less serious. Whatever blocks there are in the file are unavailable, but no further damage will occur if nothing is done. A *clri* followed by a *check −s* will restore the lost space at leisure.

In general, values of *delta* equal to or somewhat above 0, 100, or 200 are relatively innocuous; just below these numbers there is danger of spreading infection.

BUGS

Check −l or *−u* on large file systems takes a great deal of core.
Since *check* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.
It believes even preposterous super-blocks and consequently can get core images.

NAME

chown − change owner

SYNOPSIS

chown owner file ...

DESCRIPTION

The user-ID of the files is changed to *owner.* The owner may be either a decimal User-ID (UID) or a login name found in the password file.

Only the super-user is allowed to change the owner of a file, in order to do reliable disk accounting.

FILES

/etc/passwd

CLRI(VIII)

NAME

clri — clear i-node

SYNOPSIS

clri i-number filesystem

DESCRIPTION

Clri writes zeros on the 32 bytes occupied by the i-node numbered *i-number.* If the *file system* argument is given, the i-node resides on the given device, otherwise on a default file system. The file system argument must be a special file name referring to a device containing a file system. After *clri,* any blocks in the affected file will show up as "missing" in an *icheck* of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

BUGS

Whatever the default file system is, it is likely to be wrong. Specify the file system explicitly.

If the file is open, *clri* is likely to be ineffective.

NAME

crash – what to do when the system crashes

DESCRIPTION

This section gives at least a few clues about how to proceed if the system crashes. It can't pretend to be complete.

How to bring it back up. If the reason for the crash is not evident (see below for guidance on 'evident'), you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready on drive 0, stop the machine, load address 44, and start. This should write a copy of all of core on the tape with an EOF mark. Caution: Any error is taken to mean the end of core has been reached. This means that you must be sure the ring is in, the tape is ready, and the tape is clean and new. If the dump fails, you can try again, but some of the registers will be lost. See below for what to do with the tape.

In restarting after a crash, always bring up the system single-user. This is accomplished by following the directions in *boot procedures* (VIII) as modified for your particular installation; a single-user system is indicated by having a particular value in the switches (7 unless you've changed *init)* as the system starts executing. When it is running, perform a *check* or *icheck* (VIII) on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user. This is most easily accomplished by changing the single-user value in the switches to something else, then logging out by typing an EOT.

To boot UNIX at all, three files (and the directories leading to them) must be intact. First, the initialization program */etc/init* must be present and executable. If it is not, the CPU will loop in user mode at location 6. For *init* to work correctly, */dev/ln00* and */bin/sh* must be present. If either does not exist, *init* will loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, a runnable system must be obtained from a backup medium. The root file system may then be doctored as a mounted file system as described below. If there are any problems with the root file system, it is probably prudent to go to a backup system to avoid working on a running file system.

Repairing disks. The first rule to keep in mind is that an addled disk should be treated gently; it shouldn't be mounted unless necessary. If it is very valuable, yet in bad shape, it should be dumped before trying surgery. This is an area where experience and informed courage count for much.

The problems reported by *icheck* typically fall into two kinds. There can be problems with the free list: duplicates in the free list, or free blocks also in files. These can be cured easily with an *icheck −s.* There can also be problems if the same block appears in more than one file or if a file contains bad blocks, the files should be deleted, and the free list reconstructed. The best way to delete such a file is to use *clri* (VIII), then remove its directory entries. If any of the affected files is really precious, you can try to copy it to another device first.

Icheck may report files which have more directory entries than links. Such situations are potentially dangerous; *clri* discusses a special case of the problem. All the directory entries for the file should be removed. If on the other hand there are more links than directory entries, there is no danger of spreading infection, but merely some disk space that is lost for use. It is sufficient to copy the file (if it has any entries and is useful); then use *clri* on its inode and remove any directory entries that do exist.

Finally, there may be inodes reported by *check* or *icheck* that have 0 links and 0 entries. These occur on the root device when the system is stopped with pipes open, and on other file systems when the system stops with files that have been deleted while still open. A *clri* will free the inode, and an *icheck -s* will recover any missing blocks.

Why did it crash? UNIX types a message on the console typewriter when it voluntarily crashes. Here is the current list of such messages, with enough information to provide a hope at least of the remedy. The message has the form 'panic: ...', possibly accompanied by other information. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

blkdev

The *getblk* routine was called with a nonexistent major device as argument. Definitely hardware or software error.

devtab

Null device table entry for the major device used as argument to *getblk.* Definitely hardware or software error.

iinit

An I/O error reading the super-block for the root file system during initialization.

IO err in swap

An unrecoverable I/O error during a swap. Really shouldn't be a panic, but it is hard to fix.

no clock

During initialization, neither the line nor programmable clock was found to exist.

no fs

A device has disappeared from the mounted-device table. Definitely hardware or software error.

no imt

Like 'no fs', but produced elsewhere.

no procs

The system process table ran out of entries. This "can not happen" since at this point, such occurrances have already been eliminated in less dramatic ways.

out of swap space

A program needs to be swapped out, and there is no more swap space. It has to be increased. This really shouldn't be a panic, but there is no easy fix.

parity

A fatal memory parity error occurred. On 11/70's the panic happens only if the error is in kernel space.

running a dead proc

A process that has already terminated is made runnable. This "can not happen".

timeout table overflows

The system callout table ran out of entries. Try increasing NCALL in param.h.

trap

An unexpected trap has occurred within the system. This is accompanied by three numbers: a 'ka6', which is the contents of the segmentation register for the area in which the system's stack is kept; 'aps', which is the location where the hardware stored the program status word during the trap; and a 'trap type' which encodes which trap occurred. The trap types are:

0	bus error
1	illegal instruction
2	BPT/trace
3	IOT
4	power fail
5	EMT
6	recursive system call (TRAP instruction)
7	11/70 cache parity, or programmed interrupt
10	floating point trap
11	segmentation violation

In some of these cases it is possible for octal 20 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. If you wish to examine the stack after such a trap, either dump the system, or use the console switches to examine core; the required address mapping is described below.

Interpreting dumps. All file system problems should be taken care of before attempting to look at dumps. The dump should be read into the file */usr/sys/core; cp* (I) will do. At this point, you should execute *ps −alx* and *who* to print the process table and the users who were on at the time of the crash. You should dump (*od* (I)) the first 30 bytes of */usr/sys/core.* Starting at location 4, the registers R0, R1, R2, R3, R4, R5, SP and KDSA6 (KISA6 for 11/40s) are stored. If the dump had to be restarted, R0 will not be correct. Next, take the value of KA6 (location 22(8) in the dump) multiplied by 100(8) and dump 1000(8) bytes starting from there. This is the per-process data associated with the process running at the time of the crash. Relabel the addresses 140000 to 141776. R5 is C's frame or display pointer. Stored at (R5) is the old R5 pointing to the previous stack frame. At (R5)+2 is the saved PC of the calling procedure. Trace this calling chain until you obtain an R5 value of 141756, which is where the user's R5 is stored. If the chain is broken, you have to look for a plausible R5, PC pair and continue from there. Each PC should be looked up in the system's name list, using *adb* (I) and its ':' command to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible here.

SEE ALSO

clri, icheck, check on boot procedures (VIII)
"Explanation of Abnormal Conditions within the UNIX Operating System",MMF,3/17/75.

BUGS

DCHECK(VIII) DCHECK(VIII)

NAME

dcheck − file system directory consistency check

SYNOPSIS

dcheck [−inumbers] [filesystem]

DESCRIPTION

Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **−i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

FILES

Currently, /dev/rrk2 and /dev/rrp0 are the default file systems.

DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

SEE ALSO

icheck (VIII), file system (V), clri (VIII), ncheck (VIII)

BUGS

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

NAME

df − disk free

SYNOPSIS

df [filesystem]

DESCRIPTION

Df prints out the number of free blocks available on a file system. If the file system is unspecified, the free space on all of the normally mounted file systems is printed.

FILES

/dev/rf?, /dev/rk?, /dev/rp?

SEE ALSO

icheck (VIII), check (VIII)

BUGS

NAME

dump – incremental file system dump

SYNOPSIS

dump [key [arguments] filesystem]

DESCRIPTION

Dump makes an incremental file system dump on magtape of all files changed after a certain date. The *key* argument specifies the date and other options about the dump. *Key* consists of characters from the set **abcfiu0hdspnq**.

a	Normally files larger than 1000 blocks are not incrementally dumped; this flag forces them to be dumped.
b	The next argument is taken to be the maximum size of the dump tape in blocks (see **s**, **n** and **p**).
c	If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)
f	Place the dump on the next argument file instead of the tape. The default is /dev/rmt0.
i	the dump date is taken from the entry in the file /etc/dtab corresponding to the last time this file system was dumped with the **-u** option.
u	the date just prior to this dump is written on /etc/dtab upon successful completion of this dump. This file contains a date for every file system dumped with this option.
0	the dump date is taken as the epoch (beginning of time). Thus this option causes an entire file system dump to be taken.
h	the dump date is some number of hours before the current date. The number of hours is taken from the next argument in *arguments.*
d	the dump date is some number of days before the current date. The number of days is taken from the next argument in *arguments.*
s	the size of the dump tape is specified in feet. The number of feet is taken from the next argument in *arguments.* It is assumed that there are 16 standard UNIX blocks per foot at 800 *bpi* and 31 blocks per foot at 1600 *bpi* when there are 10 blocks per record. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2200 feet.
n	the size of the record(in 512-byte blocks) is taken from the next argument in *arguments.* The default size is 10 blocks per record(5120 bytes).
p	the size of the record is one block and the tape drive is /dev/mt0 unless the **f** option is used.
q	1600 *bpi* mode, tape drive /dev/rmt2 is specified unless the **f** option is used. The **q** option allows the tape size and number of reels to be calculated correctly.

If no arguments are given, the *key* is assumed to be **i** and the file system is assumed to be /dev/rp0.

Full dumps should be taken on quiet file systems as follows:

```
dump 0u /dev/rp0
check -l /dev/rp0
```

The *check* will come in handy in case it is necessary to restore individual files from this dump. Incremental dumps should then be taken when desired by:

```
dump i /dev/rp0
```

When the incremental dumps get cumbersome, a new complete dump should be taken. In this way, a restore requires loading of the complete dump tape and only the latest incremental tape.

DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a newline when this has been done. If the first block on the new tape is not writable, e.g., because you forgot the write ring, you get a chance to fix it. Generally, however, read or write failures are fatal.

FILES

/dev/rmt0	magtape
/dev/rp0	default file system
/etc/dtab	comment file

SEE ALSO

restor (VIII), check (VIII), dump (V)

BUGS

If dump tells you that it needs 1.00 tapes it may actually need 1.001 tapes(two reels). You can check for this by comparing the tape size times the record size with the number of blocks being dumped. If you wish to increase the size of the tape do so with the s option and specify more than 2200 feet.

NAME

errdemon – error logging daemon

SYNOPSIS

/etc/errdemon [file]

DESCRIPTION

The error logging daemon *errdemon* activates error logging, collects error messages (via *msg* (II)) from the system as they occur, and writes them in the file *file*. If *file* is not specified, the default file /usr/adm/errfile is used. Note that if *file* does not exist, it is created; if it already exists, error records are appended to it, so no previous error data is lost. No analysis of the error records is done by *errdemon;* that responsibility is left with *errpt* (VIII). Error logging is terminated by sending *errdemon* a software kill signal (signal 15). Only the super-user may start the error logging daemon; only one daemon may be active at any one time.

FILES

/usr/adm/errfile

SEE ALSO

errlog (II), errpt (VIII), kill (I), msg (II)

BUGS

NAME

errpt - process a report of logged errors

SYNOPSIS

errpt [file ...]

DESCRIPTION

Errpt processes data collected by the error logging mechanism (*errlog*(II) and *errdemon*(VIII)) and generates a detailed report of the contents. If no files are specified, *errpt* attempts to use */usr/adm/errfile* as *file.*

Records are processed in the order in which they appear in the input file. Startup and shutdown records frame an error logging session; time changes (via *date* (I)) occurring during that period are also noted when they occur.

Each error logging session is described with respect to time, machine configuration and a system profile. For each block device error record, the following information is presented: physical unit, logical unit, device address, retry information, simultaneous bus activity, contents of registers at the time of error, physical buffer start address, number of bytes being transferred, logical block number, and the I/O type (buffered or raw). Statistics on device activity up to the time of error is reported. For disk devices, cylinder/track/sector information is printed.

FILES

/usr/adm/errfile

SEE ALSO

errlog(II),errdemon(VIII)

BUGS

No attempt is made to recover from a bad record. Only block devices are currently handled by *errpt.* This version is unable to produce a summary report or to select particular devices for reporting.

NAME

getty — set terminal modes, speed and line discipline

SYNOPSIS

/etc/getty lnid [speed [discip]]

DESCRIPTION

Overview *getty* is a program that is invoked by *init* (VIII). It is the first process in a series *(init-getty-login-shell)* that ultimately connects a terminal user with UNIX. The lines for which a *getty* process has to be created are found in the file */etc/lines* (V). *getty* opens the terminal special file */dev/lnid* for standard input and output (file descriptors 0 and 1), reads the user's login name, and invokes the *login* (I) command with the user's name as argument. While reading the name, *getty* attempts to adapt the system to the speed and type of terminal being used.

Case Mapping and Carriage Return Treatment The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is non-empty, the system is told to map any future upper-case characters into the corresponding lower-case characters. If the user's name is terminated by a carriage-return rather than a new-line character, the system will treat carriage returns like new-line characters (see *stty* (II)).

Arguments

lnid	Identifies a special file in directory */dev*, the port of a terminal line. Conventional naming is **ln***id*, where *id* is a one or two character field.
speed	A character from [- 0 1 2 3 4] that determines what assumptions are made about the speed of the terminal connecting to this port (see below). Default is 0.
discip	A character from the set [0 1] that determines what line discipline is used. Serves internally as an index to the expandable line discipline switch table in conf.c. Default is 0.

The following table summarizes the currently recognized argument combinations.

Representative	*speed*	*discip*	**baud**	**mode**
Default†	0	0	†	FD
TTY 33	–	0	110	FD
TTY 37	1	0	150	FD
(nullmodem only)††	2	0	1200	FD
(nullmodem only)	3	0	9600	FD
TTY 40/1	4	1	1200	HD

† Any asynchronous full-duplex (FD) terminal at 300, 150, or 110 baud. See the following paragraph.

†† To be used with the 212A Data Set when available and appropriate 1200 baud terminals.

Default Operation: Speed Detection *getty* sets the speed of the interface to 300 baud, specifies raw mode (break on every character), echo suppression, tab expansion on output, and allows either parity. It types the "login:" message, which includes the characters which put the Terminet 300 terminal into full-duplex and return the GSI terminal to non-graphic mode. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the "break" ("interrupt") key, meaning that the assumed speed was wrong. Subsequent ''breaks'' will cause the terminal to cycle through the three speeds from 300 to 150 to 110 baud with the ''login'' typed for each occurrence. (If the terminal speed switch is not correspondingly set (300, 150, 110), the output will prove incomprehensible.) On a subsequent EOT (control D), the terminal speed reverts to the initialized speed (300).

Fixed Port Speeds With arguments of '-', '1', and '2' the terminal speeds are set to 110, 150, and 1200 baud, respectively. '4' sets mode and speed for 1200 baud operation of the half-duplex Teletype Model 40. (See *hd* (IV) and also *How to get started* at the beginning of this manual.) The aforementioned arguments, however, do not have the cycling ability that '0' possesses. For speeds other than the above, *getty* must be accordingly modified.

SEE ALSO

login (I), stty (II), hd (IV), lines (V), hd (IV), tty(IV) init (VIII)

BUGS

GLOB (VIII)

NAME

glob – generate command arguments

SYNOPSIS

/etc/glob command [arguments]

DESCRIPTION

Glob is used to expand arguments to the shell containing "*", "[", or "?". It is passed the argument list containing the metacharacters; *glob* expands the list and calls the indicated command. The actions of *glob* are detailed in the Shell writeup.

SEE ALSO

sh (I)

BUGS

ICHECK(VIII) ICHECK(VIII)

NAME

icheck – file system consistency check and interactive repair

SYNOPSIS

icheck [**–syn**] [filesystem] ...

DESCRIPTION

Icheck audits UNIX file systems for consistency and corrects any discrepancies. Since these corrections will, in general, result in a loss of data, the program will request operator concurrence for each such action. All questions should be answered by typing "yes" or "no", followed by a new-line character. Typing "yes" will cause the correction to take place. However, if the program does not have write permission on the file system or the "no" option, **-n,** is on, then all questions will automatically be answered "no". Alternatively the "yes" option, **-y,** will cause all questions to be answered "yes".

The program consists of six separate phases. Some phases are skipped if they are not needed. In phase one, **icheck** examines all block pointers in all files; checking for pointers which are outside of the file system (BAD) and for blocks which appear in more than one file (DUP). A table is made of all DUP blocks and all defective files are marked for clearing. Each error is printed, but no correction takes place in this phase.

The second phase is run only if DUP blocks were found in phase one. This phase finds the rest of the DUP blocks, marking each for clearing.

The third phase checks the directory structure of the file system. This is done by descending the directory tree, examining each entry. A count is kept of the number of references to each file. If any entry refers to an unallocated file, a file marked for clearing, or a file number outside the file system, then the entry is printed, and, if the operator agrees, it is removed. Refusing to remove an entry to a marked file will clear the mark, preserving the file and its subsequent entries.

In phase four all marked or unreferenced files are listed. With concurrence from the operator, each of these files is then cleared. In addition, any file whose link count does not agree with the number of references is listed; and, if agreed, the link count is adjusted.

If the salvage option, **-s,** is on, then phase five is skipped. Otherwise, check examines the free list. If any blocks are found which are outside the file system or which have been previously encountered in a file or elsewhere in the free list, then the list is pronounced BAD and a salvage is called for. Operator agreement will set the salvage option and proceed to the next phase. If there are no defects in the free list and all blocks are accounted for, the check is finished. Otherwise, the number of missing blocks (i.e. in neither a file nor the free list) is printed and a salvage is requested.

The last phase is the salvage operation; where the free list is recreated. It is run whenever the salvage option is on or a problem has been found with the free list. Simply stated, a new free list is constructed containing all blocks not found in some file.

The system responses are in general self-explanatory and follow the sequence described above. In the specification that follows, the following notation will be used:

<b>	block number
<i>	inode number
<fname>	file pathname
<n>	positive integer
<c>	option character

Icheck begins with the following output:

```
<filesystem>{(NO WRITE)}
```

Phase 1 - Check Blocks

The "(NO WRITE)" message indicates that the program does not have write permission on the file system. Therefore, subsequent corrections will be suppressed, by automatically answering "no" to all questions. Phase one then proceeds to list any BAD or DUP blocks and their inode number, as follows:

```
<b>   BAD               I=<i>
<b>   DUP               I=<i>
<b>   EXCESSIVE DUPS    I=<i>
```

If too many DUPs are encountered, the program will list all blocks. But it will not mark the excess DUPs for later processing. When Phase 1 is finished, if any DUPs were encountered, then Phase 2 is run. Otherwise, Phase 2 is skipped. This phase will list the rest of the DUP blocks as follows:

Phase 2 - Rescan for more DUPS

```
<b>   DUP               I = <i>
```

Check now descends the directory tree, asking to remove any defective entries.

Phase 3 - Check Pathnames

```
I OUT OF RANGE      I = <i>     <fname>     REMOVE?
UNALLOCATED         I = <i>     <fname>     REMOVE?
BAD/DUP             I = <i>     <fname>     REMOVE?
```

If no option has been specified, the program will wait for a response of "yes" or "no" after each question. Note, a "no" answer to the BAD/DUP entry will unmark that inode for clearing. This will suppress any subsequent correction to that file.

Now **icheck** will clear or adjust any defective files. Again, if no option has been specified, it will wait for a "yes" or "no" response to each question. The program will also indicate whether each entry is a file or a directory.

Phase 4 - Check Reference Counts

```
UNREFERENCED     {FILE/DIRECTORY}       I = <i>      CLEAR?
BAD/DUP          {FILE/DIRECTORY}       I = <i>      CLEAR?
LINK COUNT       {FILE/DIRECTORY}       I = <i>      ADJUST?
```

If the salvage option is not on, the program will now validate the free list. Otherwise, this phase is skipped. If there are any errors in the free list, it will specify them and request a salvage.

Phase 5 - Check Free List

```
BAD FREE LIST          SALVAGE?
<n> MISSING            SALVAGE?
```

Phase 6 is the salvage operation. It is only done if one has been requested.

Phase 6 - Salvage Free List

Finally, some totals are printed: the total number of allocated files (including directories and special files); the number of blocks in use; and the number of blocks in the free list.

```
<n> FILES              <n> BLOCKS           <n> FREE
```

If the file system has been modified, then the following message is printed and the program goes into a loop. This is only a reminder to the operator since the program can be forced to terminate with a <DEL> character.

```
*****BOOT UNIX(NO SYNC!)*****
```

A number of errors can terminate **icheck.** An illegal option or the inability to open the file system are shown as:

```
<c> OPTION??
CAN NOT OPEN <filesystem>
```

An I/O error on the filesystem will also cause an error message. In this case, the operator is given the choice of exiting ("yes") or continuing ("no"). This error is generally a hardware error, and continuing is rarely a good idea.

```
CAN NOT READ <filesystem>                   BLOCK <b>EXIT?
CAN NOT SEEK <filesystem>                   BLOCK <b>EXIT?
CAN NOT WRITE <filesystem>                  BLOCK <b>EXIT?
```

FILES

/dev/rootdev	default file system to be checked

BUGS

Icheck has been known to produce core images on large file systems.

NAME

init — process control initialization

SYNOPSIS

/etc/init

DESCRIPTION

Overview The primary role of *init* is to create a process for each terminal port on which a user terminal may log in. It is invoked as the last step of the UNIX boot procedure to become the ancestor of all processes except the Scheduler. Initially, and whenever one of its child processes terminates, it scans the */etc/lines* file. Depending on some logic described below, it may or may not spawn a process for an entry. The command *telinit* will also force *init* to rescan */etc/lines.*

Processing of the Lines File The entries of the */etc/lines* (V) file have the format

id:runstate:action:shcommand:process

For each entry, the parent *init* first determines whether to process the entry at all by comparing its *runstate* with the system's run state (see below), and considering the *action* requested. (The legal *action* keywords are explained in *lines* (V)). If the entry is selected for processing, it gets its optional *shcommand* executed. Then the *process* is spawned, usually */etc/getty,* which opens the line identified by *id* (see *getty(VIII)*). In a multi-user configuration, the */etc/lines* file is usually set up so that *init* will create an */etc/getty* process for each terminal port. This process opens the terminal for reading and writing with the file descriptors 0 and 1, the standard input and output, respectively. The open is not completed until someone dials up and carrier is established on the channel. *Getty* tries to determine the terminal speed, reads the user's name, and invokes *login* (I), which in turn invokes the Shell.

Run States There are 8 system run states denoted by the digits 0 - 7. Entries in the *lines* file may have any subset of the set [0-6] specified as their entry *runstate,* e.g. 01, 234, 6540. The empty set is equivalent to 0123456. System *runstate* 7 is defined as single-user state, in which case the *lines* file is ignored by *init.* By convention, *run state* 0 should be indicated in the entries of all ports and daemons in the standard multiuser configuration.

Init determines the system *run state* either from the console switches or from the argument of the *telinit* command or both. A *run state* can be viewed as a software configuration defining a selected group of (terminal and other) processes. The set of processes spawned by *init* that characterizes each of these *run states* is defined in the *lines* file.

Init at Boot Time As the last step in the boot procedure, *init* checks the console switches. If they contain a number from 0 through 7, *init* enters the corresponding system *run state.* Any number greater than 7 is treated as 0.

Single User State To enter single-user state at other than boot time, set the console switches to 7 and type *telinit S.* In system *run state* 7, the *lines* file is not processed. Rather, the console terminal, */dev/ln00,* is opened for reading and writing, and the Shell is invoked immediately. To exit from the single-user *run state,* first set the console switches to the desired *run state,* [0-6]; then, terminate the Shell with EOT (control d). Alternatively, use the *telinit* command to switch *init* to a different *runstate.* Change the console switches now, otherwise you'll leave the system vulnerable to remote superusers issuing *telinit S,* thereby enabling only the local console terminal.

Changing the Run State The *run state* can be changed by typing *telinit* d [$0 \leqslant d \leqslant 6$], independent of the console switch settings; *init* will wake up and send the warning signal (signal

15) to all those of its child processes that are undefined in the target *run state.* Then *init* waits 20 seconds before forcibly terminating these processes via the kill signal (signal 9). (Currently, most UNIX commands terminate upon receiving this signal.)

Init Bookkeeping *init* maintains a record of system events since the last boot and of currently active descendant processes in file */etc/utmp* (V), which is scratched on every boot. Similar information across boots is recorded in the (optional) history file */etc/wtmp* (V). Whenever *init* creates a process for an entry in the *lines* file, it writes a time-stamped entry into the files */etc/utmp* (V) and */etc/wtmp* (V). Whenever it learns about the termination of this process, it removes this entry from */etc/utmp* and puts a history entry into */etc/wtmp. init* also records the following special system events, which are identified by the parenthesized id field: (!B) the system was booted; (RL) the run state was changed; (OT) the old time of day before it was changed; (NT) the new time of day being set. The *who* (I) and *ac* (VIII) commands can be used to inspect or summarize these files, respectively.

Testing Init The *init* process has the ability to overlay itself with another *init* process for testing changes to *init* or for reverting to an old version of *init.* The file */etc/xinit* can be forced to overlay *init* by entering the single-user run state (see above), setting the console switches to 0177770, and typing *telinit R.*

Internals *init* maintains its own process table to keep track of its descendants (a maximum of 35, currently). Process lookup is by *id* for entry processing and by process id for terminating descendants.

FILES

/dev/ln??, /etc/utmp, /etc/wtmp, /etc/lines

SEE ALSO

login (I), sh (I), who(I), lines(V), /etc/utmp(V), /etc/wtmp(V), getty (VIII), telinit(VIII)

INO (VIII) INO (VIII)

NAME

ino − get the i-number of a file

SYNOPSIS

ino file ...

DESCRIPTION

The i-number of each file argument is printed. An i-number of zero is printed if a bad argument is given.

BUGS

IOSTAT(VIII) IOSTAT(VIII)

NAME

iostat − report I/O statistics

SYNOPSIS

iostat [**−t −s −i**] [interval [count]]

DESCRIPTION

Iostat delves into the system and reports certain statistics kept about input-output activity. Information is kept about up to different disks (RF, RK, RP, HP, HS) and about typewriters. For each disk, IO completions and number of words transferred are counted; for typewriters collectively, the number of input and output characters are counted. Also, each sixtieth of a second, the state of each disk is examined and a tally is made if the disk is active. The tally goes into one of four categories, depending on whether the system is executing in user mode, in 'nice' (background) user mode, in system mode, or idle. From all these numbers and from the known transfer rates of the devices it is possible to determine information such as the degree of IO overlap and average seek times for each device.

The optional *interval* argument causes *iostat* to report once each *interval* seconds. The first report is for all time since a reboot and each subsequent report is for the last interval only.

The optional *count* argument restricts the number of reports.

With no flag argument *iostat* reports for each disk the number of transfers per minute, the milliseconds per average seek, and the milliseconds per data transfer exclusive of seek time. It also gives the percentage of time the system has spent in each of the four categories mentioned above.

The **−t** flag causes the number of typewriter characters per second to be reported as well.

The **−i** flag requests a report of the percentage of time spend in each of the four categories mentioned above, the percentage of time each disk was active (seeking or transferring), the percentage of time any disk was active, and the percentage of time spent in 'IO wait': idle, but with a disk active.

The **−s** flag requests a printout of the raw timing information: 24 numbers indicating the percentage of time spent in each of the possible configurations of 4 system states and 8 IO states (3 disks each active or not).

FILES

/dev/mem, /unix

BUGS

LOAD(VIII) LOAD(VIII)

NAME

load − load /unix

SYNOPSIS

load [40] [45] [70]

DESCRIPTION

load is a shell file used to create a bootable unix program. The result of the execution of a *load* is an *a.out* file which is capable of being booted as the core resident UNIX. The *load* uses various libraries to build the resulting *a.out* file:

/usr/sys/lib1	Operating system library
/usr/sys/lib2	I/O library
/usr/sys/lib40	Stripped library for 11/40's
/usr/sys/conf.o	Configuration tables
/usr/sys/low.o	Low core image
/usr/sys/mch.o	Machine code

The arguments to *load* are:

40	Generates an *a.out* for 11/40 processors. The resulting *a.out* will not have separate instruction and data space since 11/40's are not capable of handling this. The size of the operating system must be smaller for the smaller processor size and for this reason messages, semaphores, error logging, and unibus mapping are removed.
45	Generates an *a.out* for 11/45 processors. The resulting *a.out* will have separate instruction and data space for the operating system and for user programs. There is no size constraint to date so all of the operating system functions are present and operational.
70	Generates an *a.out* for 11/70 processors. Currently the *a.out* produced is identical to that of the 11/45.

FILES

/usr/sys/conf.o
/usr/sys/low.o
/usr/sys/mch.o
/usr/sys/lib1
/usr/sys/lib2
/usr/source/opsys/*
/usr/source/io1/*
/usr/source/io2/*

DIAGNOSTICS

"Undefined symbol" is printed if one of the libraries are incomplete. Generally speaking this is caused by a missing driver in /usr/sys/lib2 or a missing required function in /usr/sys/lib1.

NAME

lpd − line printer daemon

SYNOPSIS

/etc/lpd

DESCRIPTION

Lpd is the line printer daemon (spool area handler) invoked by *lpr.* It uses the directory */usr/lpd.* The file *lock* in that directory is used to prevent two daemons from becoming active simultaneously. After the daemon has successfully set the lock, it scans the directory for files beginning with ''df.'' Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

L specifies that the remainder of the line is to be sent as a literal.

B specifies that the rest of the line is a file name. That file is to be sent as binary cards.

F is the same as **B** except a form feed is prepended to the file.

U specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.

FILES

/usr/lpd/*	spool area
/dev/lp	printer

SEE ALSO

lpr (I)

BUGS

Temporary files (of the form tf.*) never get removed and must be deleted manually.

NAME

mkconf – create configuration table and low core

SYNOPSIS

mkconf [**-nta**] [address]

DESCRIPTION

Mkconf creates two files, one which contains the low core vectors (l.s) and one which contains the configuration tables (c.c), needed to build a system (e.g. */unix*). It is an interactive program, which creates these two files from the input specifications supplied by the user. The specifications allow the input of the type of device, the number of devices of a given type and the vector address if the device is in the floating vector area. The root and swap devices and their minor device numbers must be specified, and the location and size of the swap area (swplo and nswap) values may be specified. If the root and swap devices are not specified the default is the *rk*. The first input to *mkconf* must be the type of processor. The following is an example of how *mkconf* is used. Notice that the devices specified as the root and swap must precede the definition of root and swap.

```
mkconf
70
hp
hs
ht
2dc
root hp 1
swap hs 8
swplo 1
nswap 4000
```

Typing *list* before hitting CTL D will produce a list of vector addresses and the devices at those locations. Typing an EOT (CTL D) terminates input and causes l.s and c.c to be generated.

Mkconf automatically adds a jump to a routine whenever there is an empty location in low core. This routine prints the message
"stray interrupt at XXX"
on the console terminal. when an interrupt occurs at one of these locations. An interrupt occurring at any of these locations is treated similarly to a device interrupt. The stray interrupt routine will print the location (modulo 128) of the interrupt, thus making it possible to narrow the number of locations down. The arguments to *mkconf* alter the use of the jump to the stray interrupt routine in the following ways.

−n	No stray interrupt vectors are produced.
−t	A jump to trap+15 is used instead of stray. This is not quite so useful but it is better than nothing.
−a	Currently, the stray interrupts are produced up to location 0400. The next argument (address) following this option can extend or shorten this area.

FILES

./c.c
/usr/sys/conf.o
/usr/source/opsys/conf.c
./l.s

/usr/sys/low.o
/usr/source/opsys/low.s

SEE ALSO

BUGS

Mkconf should have been written in yacc.

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special (proto OR filesize)

DESCRIPTION

Mkfs constructs a file system by writing on the special file *special* according to the directions found in the prototype file *proto.* The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program (see boot procedures (VIII)). The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the i-list size in blocks (remember there are 16 i-nodes per block). The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **–bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or – to specify set-user-id mode or not. The third is **g** or – for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod* (I)).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname from which the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

If the second argument is a *filesize*, *mkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *filesize* interpreted as a decimal number. The i-list size is $\frac{filesize}{43+\frac{filesize}{1000}}$ (This corresponds to an average size of three blocks per file for a 4000 block file system and six blocks per file at 40,000.) The boot program is left uninitialized.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d--777 3 1
usr     d--777 3 1
        sh      ---755 3 1 /bin/sh
        ken     d--755 6 1
                $
        b0      b--644 3 1 0 0
        c0      c--644 3 1 0 0
        $
        $
```

SEE ALSO

file system (V), directory (V), boot procedures (VIII)

BUGS

It is not possible to initialize a file larger than 64K bytes.
The size of the file system is restricted to 64K blocks.
There should be some way to specify links.
Unbalanced $'s cause unclear diagnostics.

NAME

mknod − build special file

SYNOPSIS

/etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g., unit, drive, or line number).

The assignment of major device numbers is standardized; they can be found in the system source file *conf.c.*

Mknod may only be invoked by the super-user.

SEE ALSO

mknod (II)

BUGS

MKPT(VIII) MKPT(VIII)

NAME

mkpt - make prototype file for use by mkfs

SYNOPSIS

/etc/mkpt spec proto

DESCRIPTION

Mkpt constructs a prototype file *proto* according to the directions found in the specification file *spec.* The specification file must contain the name of the boot file as the first token, the size of the file system to be created as the second token and the size of the i-list in blocks as the third token. The next set of tokens comprise the specification for the root file. See *mkfs* (VIII) for the correct format of the above tokens.

All of the following tokens consist of directory names. Directory names followed by a ':' are made directory entries in the *proto* file. Directory names not terminated by a ':' are scanned by *mkpt* and all entries are put in the *proto* file (recursively). If no ':' terminated directory name appears, all entries taken from the specified directories are entered into the *proto* file as originating from the root directory. A directory name terminated by a ':' and followed immediately by another directory name terminated by a ':' will be entered as an empty directory entry in the *proto* file. A directory name terminated by a ':' may be followed by as many directory names not terminated by ':'s as desired.

A directory name terminated by a ':' may be several levels deep, e.g. usr/usra: . All subsequent directory names terminated by ':' will be put in the directory 'usr' . One may back up a directory level by means of the entry '..:' . This enables one to extract selectively certain subdirectories of a given directory.

Editing of the *proto* file will only be required if one wishes to change the names of some files. Otherwise the *proto* file may be used exactly as is to create a new file system using *mkfs* (see *mkfs* (VIII)).

Some sample specification files follow:

```
/usr/mdec/uboot
4872 55
d--777 3 1
bin: /bin
lib: /lib
dev: /dev
etc: /etc
tmp:
usr:
mnt:
crp:
```

and

```
/usr/mdec/uboot
42000 500
d--777 3 1
/usr
```

and

```
/usr/mdec/uboot
20000 300
d--777 3 1
usra: /usr/usr1 /usr/usr2 /usr/usr3
usrb: /usr/usr4
usrc:
usrd: /usr/usr5
```

and

```
/usr/mdec/uboot
35000 450
d--777 3 1
/
```

and

```
/usr/mdec/uboot
4000 80
d--777 3 1
bin: /bin
etc: /etc
dev: /dev
lib: /lib
tmp:
usr/usra: /usr/usra
usrb: /usr/usrb
usrc:
bin: /usr/bin
mnt:
crp:
```

SEE ALSO

file system(V), directory(V), boot procedures(VIII).

DIAGNOSTICS

There are diagnostics for non-existent or non-readable directories.

BUGS

If a very large *proto* file is created, it cannot be edited to make any changes.

NAME

mount − mount file system

SYNOPSIS

/etc/mount special file [**−r**]

DESCRIPTION

Mount announces to the system that a removable file system is present on the device corresponding to special file *special* (which must refer to a disk or possibly DECtape). The *file* must exist already; it becomes the name of the root of the newly mounted file system.

Mount maintains a table of mounted devices; if invoked without an argument it prints the table.

The optional last argument indicates that the file is to be mounted "read-only". Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted. The *mount* command is reserved by the operating system to the super-user. However, if the set-user-ID bit is turned on, the command has global use.

SEE ALSO

mount (II), mtab (VII), umount (VIII)

BUGS

NCHECK(VIII) NCHECK(VIII)

NAME

ncheck – generate names from i-numbers

SYNOPSIS

ncheck [**–i** numbers] [**–u** numbers ids] [**–a**] [filesystem]

DESCRIPTION

Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. These default file systems are as follows:

/dev/rroot
/dev/rrusr
/dev/rsrc

These file systems should correspond to the root, user, and source file systems in their "raw" (character) I/O state.

The **–i** flag reduces the report to only those files whose i-numbers follow. The **–u** option reduces the report to files owned by the specified user ids. The report produced by **–u** contains the i-number, size of the file in blocks, the name of the owner of the file (if known) and the pathname of the file. All entries are separated by a tab character to facilitate sorting by size, owner, i-node or pathname. If no numbers or ids are supplied with the **–u** or **–i** option, ncheck will read numbers or ids from the standard input. The ids or numbers may be separated by tabs, blanks or newlines. **–a** allows printing of the names '.' and '..', which are ordinarily suppressed. A file system may be specified as an argument. The raw device which references the filesystem should be specified for efficiency

SEE ALSO

dcheck (VIII), icheck (VIII), sort (I)

BUGS

It is slow.

NAME

reloc − relocate object files

SYNOPSIS

reloc file octal [−]

DESCRIPTION

Reloc modifies the named object program file so that it will operate correctly at a different core origin than the one for which it was assembled or loaded.

The new core origin is the old origin increased by the given *octal* number (or decreased if the number has a '−' sign).

If the object file was generated by *ld,* the **−r** and **−d** options must have been given to preserve the relocation information and define any common symbols in the file.

If the optional last argument is given, then any *setd* instruction at the start of the file will be replaced by a no-op.

The purpose of this command is to simplify the preparation of object programs for systems which have no relocation hardware. It is hard to imagine a situation in which it would be useful to attempt directly to execute a program treated by *reloc.*

SEE ALSO

as (I), ld (I), a.out (V)

BUGS

NAME

restor − incremental file system restore

SYNOPSIS

restor key [arguments]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* argument specifies what is to be done. *Key* is a character from the set **trxwcfiq**.

t	The date that the tape was made and the date that was specified in the *dump* command are printed. A list of all of the i-numbers on the tape is also given.
r	The tape is read and loaded into the file system specified in *arguments.* This should not be done lightly (see below).
x	Each file on the tape is individually extracted into a file whose name is the file's i-number. If there are *arguments,* they are interpreted as i-numbers and only they are extracted.
c	If the tape overflows, increment the last character of its name and continue on that new drive. (Normally it asks you to change tapes.)
f	Read the dump from the next argument file instead of the tape. The default file is /dev/rmt0.
i	All read and checksum errors are reported, but will not cause termination.
w	In conjunction with the **x** option, before each file is extracted, its i-number is typed out. To extract this file, you must respond with **y** , to exit respond with **x**.
q	specifies the input file /dev/rmt2 for tapes dumped at 1600 *bpi.*

The **x** option is used to retrieve individual files. If the i-number of the desired file is not known, it can be discovered by following the file system directory search algorithm. First retrieve the *root* directory whose i-number is 1. List this file with *ls −fi 1.* This will give names and i-numbers of sub-directories. Iterating, any file may be retrieved.

The **r** option should first be used to restore an epoch dump onto a clear file system. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump. After this the file system may be brought up to date by using *restor* with the **r** option and the latest incremental dump tape.

A *dump* followed by a *mkfs* and a *restor* is used to change the size of a file system.

FILES

/dev/rmt0
/dev/rmt2

SEE ALSO

ls (I), dump (VIII), mkfs (VIII), clri (VIII)

DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor's* approach is to exit if anything is wrong.

There is also a "read error" diagnostic that is produced on systems with TU10 tape units, it is a non-fatal error and may be ignored.

NAME

sa − Shell accounting

SYNOPSIS

sa [**−abcijlmnorstuv**] [file]

DESCRIPTION

When a user logs in, if the Shell is able to open the file */usr/adm/sh_acct*, then as each command completes, the Shell writes at the end of this file the name of the command, the user, system time and real time consumed, and the user ID. *Sa* reports on, cleans up, and generally maintains this and other accounting files. To turn accounting on and off, the accounting file must be created or destroyed externally. If the user is the super-user, accounting is placed into */usr/adm/su_acct* instead. As with *sh_acct*, it must be created or destroyed externally. Similarly, if the file */usr/adm/acct* is created and system accounting is activated (using *accton* (VIII)), an accounting record is written when each process terminates.

Sa is able to condense the information in */usr/adm/su_acct* or */usr/adm/acct* into a summary file, */usr/adm/savacct*, which contains a count of the number of times each command was called and the time resources consumed. The summary file, */usr/adm/usracct*, is used to record for each user the total number of commands executed, and the total cpu time used. This condensation is desirable because on a large system accounting files can grow by 100 blocks per day. The summary file is read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; *acct* is the default. When a Shell accounting file (*sh_acct* or *su_acct*) is processed, commands that were executed from a command file have an asterisk appended to their name. If the system accounting file (*acct*) is processed, an asterisk is appended to those commands that did not do an *exec*(II) (e.g. daemons). There are many options:

a	Do not place all command names containing unprintable characters and those used only once under the name "***other."
b	Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.
c	Besides total user time, system time, and real time for each command, print percentage of total time over all commands.
i	Do not read the summary files */usr/adm/savacct* and */usr/adm/usracct*, so only information in the accounting files is condensed and reported.
j	Instead of total minutes time for each category, give seconds per call.
l	Separate system and user time; normally they are combined.
m	Superseding all other flags except **u**, print for each user the login name, number of commands executed, and the total number of minutes of cpu time used.
n	Sort by number of calls.
o	Superseding the **l** flag, report for each command the ratio of user cpu time to user + system cpu time.
r	Reverse order of sort.

s Merge accounting file into summary files */usr/adm/savacct* and */usr/adm/usracct* when done. This option also causes the **a** option to be used.

t For each command report ratio of real time to the sum of user and system times.

u Superseding all other flags, print for each command in the accounting file the day of the year, time, day of the week, user ID and command name.

v If the next character is a digit *n,* then type the name of each command used *n* times or fewer. Await a reply from the typewriter; if it begins with "y", add the command to the category "**junk**." This is used to strip out garbage.

FILES

/usr/adm/sh_acct
/usr/adm/acct
/usr/adm/savacct
/usr/adm/usracct
/usr/adm/su_acct

SEE ALSO

acct (V), ac (VIII), accton (VIII), ac (II)

BUGS

Probably.

SU (VIII) SU (VIII)

NAME

su — become privileged user

SYNOPSIS

su

DESCRIPTION

Su allows one to become the super-user, who has all sorts of marvelous (and correspondingly dangerous) powers. In order for *su* to do its magic, the user must supply a password. If the password is correct, *su* will execute the Shell with the UID set to that of the super-user. To restore normal UID privileges, type an end-of-file to the super-user Shell.

The password demanded is that of the entry ''root'' in the system's password file.

To remind the super-user of his responsibilities, the Shell substitutes '#' for its usual prompt '%'.

SEE ALSO

sh (I)

SYNC (VIII) SYNC (VIII)

NAME

sync — update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. See sync (II) for details.

SEE ALSO

sync (II)

BUGS

NAME

telinit – superuser communication with init

SYNOPSIS

telinit [arg]

DESCRIPTION

Telinit is used to direct the actions of *init* (VIII). *Telinit* can only be run by superuser. It signals *init* via the *kill* (II) system call to perform the action called for by the one-character argument *arg:*

[0-6]	Place the system in one of the run states 0-6; basically, *init* will process, on each round, only those entries in */etc/lines* which have a matching run state. If the system was in single-user state, change the console switches from 7 immediately for protection.
[a,b,c]	Process once only those */etc/lines* entries having a, b or c, respectively, in their run state field. The system run state remains unchanged. This can be used to start a single process independent of a terminal.
Q	Reexamine the */etc/lines* file; no change of system run state.
S	Switch to a single-user environment; however, the console switch register must also be set to indicate single-user (07).
R	Cause *init* to overlay itself with */etc/xinit*, if in single-user state. The console switch register must be set to **0177770**, before.

FILES

lines(V), init(VIII), kill(II), who(I), getty(VIII)

DIAGNOSTICS

Illegal Switch Settings
Illegal Argument

UMOUNT(VIII) UMOUNT(VIII)

NAME

umount − dismount file system

SYNOPSIS

/etc/umount special

DESCRIPTION

Umount announces to the system that the removable file system previously mounted on special file *special* is to be removed. Only the super-user may unmount file systems.

SEE ALSO

mount (VIII), umount (II), mtab (V)

FILES

/etc/mtab mounted device table

DIAGNOSTICS

Frequently, in the case of */usr,* the file system is found to be busy because one of the Shell accounting files (*/usr/adm/sh_acct* or */usr/adm/su_acct*) is open for some Shell or system accounting is active (*/usr/adm/acct*).

BUGS

UPDATE(VIII) UPDATE(VIII)

NAME

update — periodically update the super block

SYNOPSIS

update

DESCRIPTION

Update is a program that executes the *sync* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the /etc/lines file. See sync (II) for details.

SEE ALSO

sync (II), init (VIII)

BUGS

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync* temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.